Aug 88

User's Guide

to

GNU C++

Michael D. Tiemann

last updated 1 May 1988

for version 1.21.0

Copyright © 1988 Free Software Foundation, Inc.

The *User's Guid to GNU C++* is a derivative work of authorship based on the *Internals of GNU CC* by Richard Stallman, and documentary comments in the source code of Doug Lea's library functions. Both of these documents are copyright ©The Free Software Foundation.

The code described in this document represents a derivative work of authorship of the GNU CC compiler. The earlier GNU CC compiler was written by Richard Stallman. All copyright conditions applying to GNU CC also apply to GNU C++.

Permission is granted to make and distribute verbatim copies of this manual provided the copyright notice and this permission notice are preserved on all copies.

Permission is granted to copy and distribute modified versions of this manual under the conditions for verbatim copying, provided also that the section entitled "GNU CC General Public License" is included exactly as in the original, and provided that the entire resulting derived work is distributed under the terms of a permission notice identical to this one.

Permission is granted to copy and distribute translations of this manual into another language, under the above conditions for modified versions, except that the section entitled "GNU CC General Public License" may be included in a translation approved by the author instead of in the original English.

Note: GNU C++ is still in test release. Known bugs are documented in the "BugList" section. The "Projects" section lists things that could still be done.

Table of Contents

GNU CC GENERAL PUBLIC LICENSE 1
COPYING POLICIES 1
NO WARRANTY 3
Contributors to GNU C++ 5
1. GNU C++ Command Options 7
2. Installing GNU C++ 19
3. Trouble in Installation 23
4. GNU C++ Header Files and Libraries 25
4.1 Header Files 25
4.2 The stream class 27
4.3 The File class 28
4.3.1 Binding 28
4.3.2 Basic IO 29
4.3.3 File Control 30
4.3.4 File Status 30
4.3.5 The SFile class 31
4.3.6 The PlotFile Class 32
4.4 The String class 32
4.4.1 String Constructors 33
4.4.2 Regex constructors 34
4.4.3 examples 35
4.4.4 Matching 36
4.4.5 Substring extraction 37
4.4.6 Concatenation 38
4.4.7 String manipulation 38
4.4.8 Reading and writing 39
4.4.9 Conversion 39
4.5 The Integer class. 40
4.6 Obstacks 42
5. Incompatibilities of GNU C++ 47
6. GNU Extensions to the C++ Language 51
6.1 Statements and Declarations inside of Expressions 51
6.2 Naming an Expression's Type 52
6.3 Referring to a Type with `typeof` 52
6.4 Generalized Lvalues 53
6.5 Conditional Expressions with Omitted Middle-Operands 54

6.6 Arrays of Length Zero . . . 55
6.7 Arrays of Variable Length . . . 55
6.8 Non-Lvalue Arrays May Have Subscripts . . . 57
6.9 Arithmetic on void-Pointers and Function Pointers . . . 57
6.10 Non-Constant Initializers . . . 57
6.11 Constructor Expressions . . . 58
6.12 Dollar Signs in Identifier Names . . . 58
6.13 Inquiring about the Alignment of a Type or Variable . . . 59
6.14 Assembler Instructions with C Expression Operands . . . 59
6.15 Controlling Names Used in Assembler Code . . . 61
6.16 Non-C operators: operator <? and operator >? . . . 62
6.17 Controlling operator new . . . 63
6.18 Function calls as first-class objects . . . 63

7. Features of GNU C++ . . . 67
7.1 Structures as Arguments . . . 67
7.2 The goto statement in GNU C++ . . . 67
7.3 Type instantiation in GNU C++ . . . 67
7.4 Incremetal Linking and GNU C++ . . . 69

8. Reporting Bugs . . . 71
8.1 Have You Found a Bug? . . . 71
8.2 How to Report Bugs . . . 72

9. GNU C++ and Portability . . . 75

10. Interfacing to GNU C++ Output . . . 77

11. Passes and Files of the Compiler . . . 79

12. The Configuration File . . . 85

13. Things still left to do for GNU C++ . . . 87

14. List of currently known bugs in GNU C++ . . . 89

15. Related documentation and bibliography . . . 91

GNU CC GENERAL PUBLIC LICENSE

(Clarified 11 Feb 1988)

The license agreements of most software companies keep you at the mercy of those companies. By contrast, our general public license is intended to give everyone the right to share GNU CC. To make sure that you get the rights we want you to have, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. Hence this license agreement.

Specifically, we want to make sure that you have the right to give away copies of GNU CC, that you receive source code or else can get it if you want it, that you can change GNU CC or use pieces of it in new free programs, and that you know you can do these things.

To make sure that everyone has such rights, we have to forbid you to deprive anyone else of these rights. For example, if you distribute copies of GNU CC, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must tell them their rights.

Also, for our own protection, we must make certain that everyone finds out that there is no warranty for GNU CC. If GNU CC is modified by someone else and passed on, we want its recipients to know that what they have is not what we distributed, so that any problems introduced by others will not reflect on our reputation.

Therefore we (Richard Stallman and the Free Software Foundation, Inc.) make the following terms which say what you must do to be allowed to distribute or change GNU CC.

COPYING POLICIES

1. You may copy and distribute verbatim copies of GNU CC source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy a valid copyright notice "Copyright © 1988 Free Software Foundation, Inc." (or with whatever year is appropriate); keep intact the notices on all files that refer to this License Agreement and to the absence of any warranty; and give any other recipients of the GNU CC program a copy of this License Agreement along with the program. You may charge a distribution fee for the physical act of transferring a copy.
2. You may modify your copy or copies of GNU CC or any portion of it, and copy and distribute such modifications under the terms of Paragraph 1 above, provided that you also do the following:

- cause the modified files to carry prominent notices stating that you changed the files and the date of any change; and
- cause the whole of any work that you distribute or publish, that in whole or in part contains or is a derivative of GNU CC or any part thereof, to be licensed at no charge to all third parties on terms identical to those contained in this License Agreement (except that you may choose to grant more extensive warranty protection to some or all third parties, at your option).
- You may charge a distribution fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

Mere aggregation of another unrelated program with this program (or its derivative) on a volume of a storage or distribution medium does not bring the other program under the scope of these terms.

3. You may copy and distribute GNU CC (or any portion of it in under Paragraph 2) in object code or executable form under the terms of Paragraphs 1 and 2 above provided that you also do one of the following:
   - accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Paragraphs 1 and 2 above; or,
   - accompany it with a written offer, valid for at least three years, to give any third party free (except for a nominal shipping charge) a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Paragraphs 1 and 2 above; or,
   - accompany it with the information you received as to where the corresponding source code may be obtained. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form alone.)

   For an executable file, complete source code means all the source code for all modules it contains; but, as a special exception, it need not include source code for modules which are standard libraries that accompany the operating system on which the executable file runs.
4. You may not copy, sublicense, distribute or transfer GNU CC except as expressly provided under this License Agreement. Any attempt otherwise to copy, sublicense, distribute or transfer GNU CC is void and your rights to use the program under this License agreement shall be automatically terminated. However, parties who have received computer software programs from you with this License Agreement will not have their licenses terminated so long as such parties remain in full compliance.
5. If you wish to incorporate parts of GNU CC into other free programs whose distribution conditions are different, write to the Free Software Foundation at 675 Mass Ave, Cambridge, MA 02139. We have not yet worked out a simple rule that can be stated here, but we will often permit this. We will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software.

Your comments and suggestions about our licensing policies and our software are welcome! Please contact the Free Software Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, or call (617) 876-3296.

NO WARRANTY

BECAUSE GNU CC IS LICENSED FREE OF CHARGE, WE PROVIDE ABSOLUTELY NO WARRANTY, TO THE EXTENT PERMITTED BY APPLICABLE STATE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING, FREE SOFTWARE FOUNDATION, INC, RICHARD M. STALLMAN AND/OR OTHER PARTIES PROVIDE GNU CC "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF GNU CC IS WITH YOU. SHOULD GNU CC PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW WILL RICHARD M. STALLMAN, THE FREE SOFTWARE FOUNDATION, INC., AND/OR ANY OTHER PARTY WHO MAY MODIFY AND REDISTRIBUTE GNU CC AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY LOST PROFITS, LOST MONIES, OR OTHER SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS) GNU CC, EVEN IF YOU HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES, OR FOR ANY CLAIM BY ANY OTHER PARTY.

Contributors to GNU C++

Before GNU C++ was even conceived, Richard Stallman had begun his work on GNU CC, a program which would finally bring state-of-the-art compiler technology to the general public under his Free Software philosophy. GNU C++ serves to provide the same high-quality compiler technology with a front end to support the growing user base of object oriented programmers, especially those who are currently using C++.

Aside from Michael Tiemann, who worked out the front end for GNU C++, and Richard Stallman, who worked out the back end, the following people (not including those who have made their contributions to GNU CC) should not go unmentioned.

- Doug Lea contributed the GNU C++ library. This includes support for streams, obstacks, structured files, and other public service objects.

1. GNU C++ Command Options

The GNU C++ compiler uses a command syntax much like the AT&T C++ compiler. The `g++` program accepts options and file names as operands. Multiple single-letter options may *not* be grouped: '`-dr`' is very different from '`-d -r`'.

When you invoke GNU C++, it normally does preprocessing, compilation, assembly and linking. File names which end in '`.c`' are taken as GNU C++ source to be preprocessed and compiled; compiler output files plus any input files with names ending in '`.s`' are assembled; then the resulting object files, plus any other input files, are linked together to produce an executable.

Unlike C++, there is no '`-F`' option. This is because GNU C++ is a native-code C++ compiler, not a front-end pre-processor. The advantages of this organization are faster compilation speed, better error-reporting capabilities, better opportunity for compiler optimization, and true source-level debuggability with the GDB+ debugger.

Command options allow you to stop this process at an intermediate stage. For example, the '`-c`' option says not to run the linker. Then the output consists of object files output by the assembler.

Other command options are passed on to one stage. Some options control the preprocessor and others the compiler itself. Yet other options control the assembler and linker; these are not documented here because the GNU assembler and linker are not yet released.

Here are the options to control the overall compilation process, including those that say whether to link, whether to assemble, and so on.

'`-o` *file*' Place output in file *file*. This applies regardless to whatever sort of output is being produced, whether it be an executable file, an object file, an assembler file or preprocessed C code.

If '`-o`' is not specified, the default is to put an executable file in '`a.out`', the object file '*source*`.cc`' in '*source*`.o`', an assembler file in '*source*`.s`', and preprocessed C on standard output.

'`-c`' Compile or assemble the source files, but do not link. Produce object files with names made by replacing '`.cc`' or '`.s`' with '`.o`' at the end of the input file names. Do nothing at all for object files specified as input.

It is intended that the compiler driver of GNU CC will invoke the appropriate translator (or series of translators) for a given source file. Currently, the translators are selected on the basis of their file extension. So that one driver can be used for many different

translators, it is important that these extensions be distinct. It is strongly suggested that users become accustomed to using a '`.cc`' file extension for GNU C++ code, to distinguish it from the '`.c`' file extension already used for GNU CC code.

'`-S`' Compile into assembler code but do not assemble. The assembler output file name is made by replacing '`.cc`' with '`.s`' at the end of the input file name. Do nothing at all for assembler source files or object files specified as input.

'`-E`' Run only the GNU C++ preprocessor. Preprocess all the GNU C++ source files specified and output the results to standard output.

'`-v`' Compiler driver program prints the commands it executes as it runs the preprocessor, compiler proper, assembler and linker. Some of these are directed to print their own version numbers.

'`-B`*prefix*' Compiler driver program tries *prefix* as a prefix for each program it tries to run. These programs are '`cpp+`', '`c++`', '`as`' and '`ld++`'.

For each subprogram to be run, the compiler driver first tries the '`-B`' prefix, if any. If that name is not found, or if '`-B`' was not specified, the driver tries two standard prefixes, which are '`/usr/lib/gcc-`' and '`/usr/local/lib/gcc-`'. If neither of those results in a file name that is found, the unmodified program name is searched for using the directories specified in your '`PATH`' environment variable.

The run-time support file '`gnulib+`' is also searched for using the '`-B`' prefix, if needed. If it is not found there, the two standard prefixes above are tried, and that is all. The file is left out of the link if it is not found by those means. This library is necessary if any of the modules linked call constructors.

These options control the details of GNU C++ compilation itself.

'`-ansi`' Support all ANSI standard C programs, as best we can.

This turns off certain features of GNU C++ that are incompatible with ANSI C, such as the `asm` and `typeof` keywords, and predefined macros such as `unix` and `vax` to identify the type of system you are using. It also enables the undesirable and rarely used ANSI trigraph feature.

The '`-ansi`' option does not cause non-ANSI programs to be rejected gratuitously. For that, '`-pedantic`' is required in addition to '`-ansi`'.

The macro `__STRICT_ANSI__` is predefined when the '`-ansi`' option is used. Some header files may notice this macro and refrain from declaring certain functions or defining certain macros that the ANSI standard doesn't call for; this is to avoid interfering with any programs that might use these names for other things.

With this option enabled, differences between GNU C++ and AT&T C++ are also flagged. Because the C++ language definition and the ANSI draft differ on the inter-

pretation of syntactically identical constructs, it is unlikely that this flag could possibly be of any real use. (For this reason, this flag is currently not fully implemented).

'`-traditional`'

Attempt to support some aspects of traditional C compilers. Specifically:

- All `extern` declarations take effect globally even if they are written inside of a function definition. This includes implicit declarations of functions.
- The keywords `typeof`, `inline`, `signed`, `const` and `volatile` are not recognized.
- Comparisons between pointers and integers are always allowed.
- Integer types `unsigned short` and `unsigned char` promote to `unsigned int`.
- In the preprocessor, comments convert to nothing at all, rather than to a space. This allows traditional token concatenation.
- In the preprocessor, single and double quote characters are ignored when scanning macro definitions, so that macro arguments can be replaced even within a string or character constant. Quote characters are also ignored when skipping text inside a failing conditional directive.

'`-O`' Optimize. Optimizing compilation takes somewhat more time, and a lot more memory for a large function.

Without '`-O`', the compiler's goal is to reduce the cost of compilation and to make debugging produce the expected results. Statements are independent: if you stop the program with a breakpoint between statements, you can then assign a new value to any variable or change the program counter to any other statement in the function and get exactly the results you would expect from the source code.

Without '`-O`', only variables declared `register` are allocated in registers. The resulting compiled code is a little worse than produced by C++/PCC without '`-O`'.

With '`-O`', the compiler tries to reduce code size and execution time.

Some of the '`-f`' options described below turn specific kinds of optimization on or off.

'`-g`' Produce debugging information in DBX+ format. Programs compiled with this option can be debugged with GDB+ *at the C++ source language level.* Scope resolution, member functions, virtual functions, static class members, inline functions, pointers to members, etc., can be, for the first time, manipulated in as natural a fashion C debuggers handle C code. See the GDB+ documentation for more information about using this unique facility.

Unlike most other C++ compiler systems, GNU C++ allows you to use '`-g`' with '`-O`'. The shortcuts taken by optimized code may occasionally produce surprising results: some variables you declared may not exist at all; flow of control may briefly move where you did not expect it; some statements may not be executed because they compute constant results or their values were already at hand; some statements may execute in different places because they were moved out of loops. Nevertheless it proves possible

to debug optimized output. This makes it reasonable to use the optimizer for programs that might have bugs.

For the present, COFF (Common Object File For nat) is not supported.

'`-g0`' Produce debugging information in DBX format. This format is fully compatible with vanilla DBX. However, the extensions to the C language which are the essence of C++ will be inaccessible.

If you are running on a COFF system, you will be forced to use this flag until somebody makes '`-g`' work with COFF.

'`-gg`' Produce debugging information in GDB+ format.

(NOTE: GDB native format cannot be specified until the GNU assembler has been released. GDB+ is, however, DBX compatible, and also has X window support for those who like to debug in windows.) The GNU C++ compiler takes advantage of the fact that GNU C++ is likely to be running on a system which already provides GDB. For these systems, an extended front-end for GDB, called GDB+ is available which allows GNU C++ expressions to be evaluated in the expected way. Although the compiler does rename variables for its own convenience, GDB+ can display variables by the names they were given, call methods with argument lists as they appear in the source code, as well as list routines and set breakpoints by method names.

Because of the way that GDB's internal data structures work, it is highly unlikely that compatibility with raw GDB format will ever be supported. This should not be a problem, however, because if the system supports GDB, it should support GDB+ as well.

'`-w`' Inhibit all warning messages.

'`-W`' Print extra warning messages for these events:

- An automatic variable is used without first being initialized.

  These warnings are possible only in optimizing compilation, because they require data flow information that is computed only when optimizing. They occur only for variables that are candidates for register allocation. Therefore, they do not occur for a variable that is declared `volatile`, or whose address is taken, or whose size is other than 1, 2, 4 or 8 bytes. Also, they do not occur for structures, unions or arrays, even when they are in registers.

  Note that there may be no warning about a variable that is used only to compute a value that itself is never used, because such computations may be deleted by the flow analysis pass before the warnings are printed.

  These warnings are made optional because GNU C++ is not smart enough to see all the reasons why the code might be correct despite appearing to have an error. Here is one example of how this can happen:

```
{
```

```
int x;
switch (y)
  {
  case 1: x = 1;
    break;
  case 2: x = 4;
    break;
  case 3: x = 5;
    break;
  }
foo (x);
}
```

If the value of y is always 1, 2 or 3, then x is always initialized, but GNU C++ doesn't know this. Here is another common case:

```
{
  int save_y;
  if (change_y) save_y = y, y = new_y;
  ...
  if (change_y) y = save_y;
}
```

This has no bug because x is used only if it is set.

- A nonvolatile automatic variable might be changed by a call to `longjmp`. These warnings as well are possible only in optimizing compilation.

  The compiler sees only the calls to `setjmp`. It cannot know where `longjmp` will be called; in fact, a signal handler could call it at any point in the code. As a result, you may get a warning even when there is in fact no problem because `longjmp` cannot in fact be called at the place which would cause a problem.

- A function can return either with or without a value. (Falling off the end of the function body is considered returning without a value.) For example, this function would inspire such a warning:

```
foo (a)
{
  if (a > 0)
    return a;
}
```

  Spurious warnings can occur because GNU CC (and hence GNU C++) does not realize that certain functions (including `abort` and `longjmp`) will never return.

In the future, other useful warnings may also be enabled by this option.

'`-Wimplicit`'

Warn whenever a function is implicitly declared. This is the default for GNU C++. It can be turned off with the option `-fno-warn-implicit`.

'`-Wreturn-type`'

Warn whenever a function is defined with a return-type that defaults to `int`. Also warn

about any `return` statement with no return-value in a function whose return-type is not `void`. The option `-Wno-return-type` will disable it.

'`-Wcomment`'
: Warn whenever a comment-start sequence '`/*`' appears in a comment.

'`-Wall`'
: All of the above '`-W`' options combined.

'`-p`'
: **This option is not supported until the GNU profiler is available.** Generate extra code to write profile information suitable for the analysis program `prof`.

'`-pg`'
: **This option is not supported until the GNU profiler is available.** Generate extra code to write profile information suitable for the analysis program `gprof`.

'`-l`*library*'
: Search a standard list of directories for a library named *library*, which is actually a file named '`lib`*library*`.a`'. The linker uses this file as if it had been specified precisely by name.

  The directories searched include several standard system directories plus any that you specify with '`-L`'.

  Normally the files found this way are library files—archive files whose members are object files. The linker handles an archive file by through it for members which define symbols that have so far been referenced but not defined. But if the file that is found is an ordinary object file, it is linked in the usual fashion. The only difference between an '`-l`' option and the full file name of the file that is found is syntactic and the fact that several directories are searched.

'`-L`*dir*'
: Add directory *dir* to the list of directories to be searched for '`-l`'.

'`-nostdlib`'
: Don't use the standard system libraries and startup files when linking. Only the files you specify (plus '`gnulib+`') will be passed to the linker.

'`-m`*machinespec*'
: Machine-dependent option specifying something about the type of target machine. These options are defined by the macro `TARGET_SWITCHES` in the machine description. The default for the options is also defined by that macro, which enables you to change the defaults.

  These are the '`-m`' options defined in the 68000 machine description:

  '`-m68020`'
  : Generate output for a 68020 (rather than a 68000). This is the default if you use the unmodified sources.

  '`-m68000`'
  : Generate output for a 68000 (rather than a 68020).

  '`-m68881`'
  : Generate output containing 68881 instructions for floating point. This is the default if you use the unmodified sources.

  '`-msoft-float`'
  : Generate output containing library calls for floating point.

  '`-mshort`'
  : Consider type `int` to be 16 bits wide, like `short int`.

'-mnobitfield'

Do not use the bit-field instructions. '-m68000' implies '-mnobitfield'.

'-mbitfield'

Do use the bit-field instructions. '-m68020' implies '-mbitfield'. This is the default if you use the unmodified sources.

'-mrtd' Use a different function-calling convention, in which functions that take a fixed number of arguments return with the `rtd` instruction, which pops their arguments while returning. This saves one instruction in the caller since there is no need to pop the arguments there.

This calling convention is incompatible with the one normally used on Unix, so you cannot use it if you need to call libraries compiled with the Unix compiler.

Also, you must provide function prototypes for all functions that take variable numbers of arguments (including `printf`); otherwise incorrect code will be generated for calls to those functions.

In addition, seriously incorrect code will result if you call a function with too many arguments. (Normally, extra arguments are harmlessly ignored.)

The `rtd` instruction is supported by the 68010 and 68020 processors, but not by the 68000. The `rtd` instruction is also supported by the National 32032 family (`ret $n`), as well as the Intel iAPX processor family (`ret $n`). If the target machine does not support this operation, a warning message will be emitted.

These '-m' options are defined in the Vax machine description:

'-munix' Do not output certain jump instructions (`aobleq` and so on) that the Unix assembler for the Vax cannot handle across long ranges.

'-mgnu' Do output those jump instructions, on the assumption that you will assemble with the GNU assembler.

'-mg' Output code for g-format floating point numbers instead of d-format.

'-f*flag*' Specify machine-independent flags. These are the flags:

'-ffloat-store'

Do not store floating-point variables in registers. This prevents undesirable excess precision on machines such as the 68000 where the floating registers (of the 68881) keep more precision than a `double` is supposed to have.

For most programs, the excess precision does only good, but a few programs rely on the precise definition of IEEE floating point. Use '-ffloat-store' for such programs.

'-fno-asm'

Do not recognize `asm` or `typeof` as a keyword. These words may then be used as identifiers.

'`-fno-defer-pop`'

Always pop the arguments to each function call as soon as that function returns. Normally the compiler (when optimizing) lets arguments accumulate on the stack for several function calls and pops them all at once.

'`-fcombine-regs`'

Allow the combine pass to combine an instruction that copies one register into another. This might or might not produce better code when used in addition to '`-O`'. I am interested in hearing about the difference this makes.

'`-fforce-mem`'

Force memory operands to be copied into registers before doing arithmetic on them. This may produce better code by making all memory references potential common subexpressions. When they are not common subexpressions, instruction combination should eliminate the separate register-load. I am interested in hearing about the difference this makes.

'`-fforce-addr`'

Force memory address constants to be copied into registers before doing arithmetic on them. This may produce better code just as '`-fforce-mem`' may. I am interested in hearing about the difference this makes.

'`-fomit-frame-pointer`'

Don't keep the frame pointer in a register for functions that don't need one. This avoids the instructions to save, set up and restore frame pointers; it also makes an extra register available in many functions. **It also makes debugging impossible.**

On some machines, such as the Vax, this flag has no effect, because the standard calling sequence automatically handles the frame pointer and nothing is saved by pretending it doesn't exist. The macro `FRAME_POINTER_REQUIRED` controls whether a target machine supports this flag. See the `Registers` section in the "Internals of GNU CC" document for more details.

'`-finline-functions`'

Integrate all simple functions into their callers. The compiler heuristically decides which functions are simple enough to be worth integrating in this way.

If all calls to a given function are integrated, then the function is normally not output as assembler code in its own right. Note that in C++, declaring a function to be `inline` implicitly declares it to be `static` as well.

'`-fkeep-inline-functions`'

Even if all calls to a given function are integrated, nevertheless output a separate run-time callable version of the function.

'`-fwritable-strings`'

Store string constants in the writable data segment and don't uniquize them. This is for compatibility with o d programs which assume they can write into string constants. Writing into string constants is a very bad idea; "constants" should be constant.

'`-fno-function-cse`'

Do not put function addresses in registers; make each instruction that calls a constant function contain the function's address explicitly.

This option results in less efficient code, but some strange hacks that alter the assembler output may be confused by the optimizations performed when this option is not used.

'`-fvolatile`'

Consider all memory references through pointers to be volatile.

'`-funsigned-char`'

Let the type `char` be the unsigned, like `unsigned char`.

Each kind of machine has a default for what `char` should be. It is either like `unsigned char` by default or like `signed char` by default. (Actually, at present, the default is always signed.)

The type `char` is always a distinct type from either `signed char` or `unsigned char`, even though its behavior is always just like one of those two.

'`-fsigned-char`'

Let the type `char` be signed, like `signed char`.

'`-ffixed-`*reg*'

Treat the register named *reg* as a fixed register; generated code should never refer to it (except perhaps as a stack pointer, frame pointer or in some other fixed role).

reg must be the name of a register. The register names accepted are machine-specific and are defined in the `REGISTER_NAMES` macro in the machine description macro file.

'`-fcall-used-`*reg*'

Treat the register named *reg* as an allocatable register that is clobbered by function calls. It may be allocated for temporaries or variables that do not live across a call. Functions compiled this way will not save and restore the register *reg*.

Use of this flag for a register that has a fixed pervasive role in the machine's execution model, such as the stack pointer or frame pointer, will produce disastrous results.

'`-fcall-saved-`*reg*'

Treat the register named *reg* as an allocatable register saved by functions. It may be allocated even for temporaries or variables that live across a call. Functions compiled this way will save and restore the register *reg* if they use it.

Use of this flag for a register that has a fixed pervasive role in the machine's execution model, such as the stack pointer or frame pointer, will produce disastrous results.

A different sort of disaster will result from the use of this flag for a register in which function values are may be returned.

'`-fstrict-prototype`'

Consider the declaration `int foo ();`. In C++, this means that the function `foo` takes no arguments. In ANSI C, this is declared `int foo(void);`. With the flag `-fno-strict-prototype`, declaring functions with no arguments is equivalent to declaring its argument list to be untyped, i.e., `int foo ();` is equivalent to saying `int foo (...);`.

'`-fchar-charconst`'

This flag gives character constants the type `char`. This means that overloaded function which distinguishes characters from integers will, given a character constant argument (e.g. `'\n'`), treat that constant as a character rather than as an integer. This is useful when using the stream library: the statement `cout << '\n'` will print a newline on the stream `cout`, instead of the number 10.

'`-fconst-is-variable`'

The C++ interpretation of a variable declared `const` is that the variable will not change its value. Therefore, it is possible for the compiler to substitute the initial value of the variable in every place that the variable's value is referenced. Numerous C++ programs have been written which use this feature in places where an actual constant is required (such as a case label, or an array size declaration). GNU C++ supports this interpretation, which is in violation of the ANSI C interpretation. To get the effect of the ANSI interpretation, use this flag. Also, if you do use it, please send me mail (tiemann@mcc.com) and tell me *why* you are using it.

'`-flabels-ok`'

Allow labels to be found in name-space. The compiler will allow a variable to be assigned to the value of a label, thus making complex jump tables constructible within GNU C++. Note that labels are not lvalues, and cannot be assigned to variables.

'`-d`*letters*' Says to make debugging dumps at times specified by *letters*. Here are the possible letters:

'`r`' Dump after RTL generation.

'`j`' Dump after first jump optimization.

'J' Dump after last jump optimization.

's' Dump after CSE (including the jump optimization that sometimes follows CSE).

'L' Dump after loop optimization.

'f' Dump after flow analysis.

'c' Dump after instruction combination.

'l' Dump after local register allocation.

'g' Dump after global register allocation.

'm' Print statistics on memory usage, at the end of the run.

'-pedantic'

Attempt to support strict ANSI standard C. Since C++ invalidates a number of ANSI constructions, this switch is of dubious value. Some attempt has been made to warn about non-standard C++ features, however, even this is of uncertain value, as there are two C++ standards currently in existence: the standard as documented by AT&T, and the standard as implemented by the AT&T C++ compiler. Valid C++ programs should compile properly with or without this switch. However, without this switch, certain useful or traditional constructs banned by the standard are supported. With this switch, they are rejected. There is no reason to use this switch; it exists only to satisfy curious pedants.

These options control the C preprocessor, which is run on each C source file before actual compilation. If you use the '-E' option, nothing is done except C preprocessing. Some of these options make sense only together with '-E' because they request preprocessor output that is not suitable for actual compilation.

'-C' Tell the preprocessor not to discard comments. Used with the '-E' option.

'-I*dir*' Search directory *dir* for include files.

'-I-' Any directories specified with '-I' options before the '-I-' option are searched only for the case of '#include "*file*"'; they are not searched for '#include <*file*>'.

If additional directories are specified with '-I' options after the '-I-', these directories are searched for all '#include' directives. (Ordinarily *all* '-I' directories are used this way.)

In addition, the '-I-' option inhibits the use of the current directory as the first search directory for '#include "*file*"'. Therefore, the current directory is searched only if it is requested explicitly with '-I.'. Specifying both '-I-' and '-I.' allows you to control precisely which directories are searched before the current one and which are searched after.

'`-nostdinc`'

Do not search the standard system directories for header files. Only the directories you have specified with '`-I`' options (and the current directory, if appropriate) are searched.

Between '`-nostdinc`' and '`-I-`', you can eliminate all directories from the search path except those you specify.

'`-M`' Tell the preprocessor to output a rule suitable for **make** describing the dependencies of each source file. For each source file, the preprocessor outputs one **make**-rule whose target is the object file name for that source file and whose dependencies are all the files '`#include`'d in it. This rule may be a single line or may be continued with '\'-newline if it is long.

'`-M`' implies '`-E`'.

'`-MM`' Like '`-M`' but the output mentions only the user-header files included with '`#include "`*file*`"`'. System header files included with '`#include <`*file*`>`' are omitted.

'`-MM`' implies '`-E`'.

'`-D`*macro*' Define macro *macro* with the empty string as its definition.

'`-D`*macro*`=`*defn*'

Define macro *macro* as *defn*.

'`-U`*macro*' Undefine macro *macro*.

'`-T`' Support ANSI C trigraphs. You don't want to know about this brain-damage. The '`-ansi`' option also has this effect.

2. Installing GNU C++

1. Edit '`Makefile`'. If you are using HPUX, you must make a few changes described in comments at the beginning of the file.
2. Choose configuration files.
   - Make a symbolic link named '`config.h`' to the top-level config file for the machine you are using (see chapter 12 [Config], page 85). This file is responsible for defining information about the host machine. It includes '`tm.h`'.

     The file's name should be '`config-`*machine*`.h`'. On VMS, use '`config-vms.h`' rather than '`config-vax.h`'. On the HP 9000 series 300, use '`config-hp9k3.h`' rather than '`config-m68k.h`'.

     If your system does not support symbolic links, you might want to set up '`config.h`' to contain a '`#include`' command which refers to the appropriate file.
   - Make a symbolic link named '`tm.h`' to the machine-description macro file for your machine (its name should be '`tm-`*machine*`.h`').

     For the 68000/68020, do not use '`tm-m68k.h`' directly; instead use one of the files '`tm-sun3.h`', '`tm-sun2.h`', '`tm-isi68.h`', '`tm-news800.h`' or '`tm-3b1.h`'. Each of those files includes '`tm-m68k.h`' but sets up a few things differently as appropriate to the specific model of machine.

     There are two files you can use for a 680x0 running HPUX: '`tm-hp9k320.h`' and '`tm-hp9k320g.h`'. Use the former if you are installing GNU CC alone. The latter is for another option where GNU CC together with the GNU assembler, linker, debugger and other utilities are used to replace all of HPUX that deals with compilation. Not all of the pieces of GNU software needed for this mode of operation are as yet in distribution; full instructions will appear here in the future.

     For the 32000, use '`tm-sequent.h`' if you are using a Sequent machine, or '`tm-encore.h`' for an Encore machine; otherwise, perhaps '`tm-ns32k.h`' will work for you.

     Note: the modified GNU linker which is distributed with GNU C++ does not yet work on the Sequent. This is because it uses non-standard '`a.out.h`' format (in order to handle shared vs. private text and data). When building the compiler driver '`g++`', be sure to define `NO_GNU_LD`.

     For the vax, use '`tm-vax.h`' on BSD Unix. VMS is not yet supported.
   - Make a symbolic link named '`md`' to the machine description pattern file (its name should be '*machine*`.md`').
   - Make a symbolic link named '`aux-output.c`' to the output subroutine file for your machine (its name should be '`output-`*machine*`.c`').
3. Make sure the Bison parser generator is installed. (This is unnecessary if the Bison output file '`parse.tab.c`' is more recent than '`parse.y`' and you do not plan to change '`parse.y`'.)

Note that if you have an old version of Bison you may get an error from the line with the '`%expect`' directive. If so, simply remove that line from '`parse.y`' and proceed.

The C++ grammar is inherently ambiguous. Given enough left and right context, a recursive-descent parser will often be able to guess the user's intentions when analyzing a piece of code. GNU C++ is implemented using a simple LALR parser which does not have backup and restart capabilities. As a result, it cannot handle some of the harder cases of C++ syntax. Fortunately, the real problems only occur when trying to maintain backwards compatability. When making the GNU C++ parser you will notice a message from BISON (or YACC) that the grammar contains reduce/reduce conflicts. For now, that is the way it is. For more details, see the Projects (See chapter 13 [Projects], page 87) section.

4. If you are using a Sun, make sure the environment variable `FLOAT_OPTION` is not set. If this option were set to `f68881` when '`gnulib`' is compiled, the resulting code would demand to be linked with a special startup file and will not link properly without special pains.
5. Build the compiler. Just type '`make`' in the compiler directory.

   The compiler you have just built is now ready to run. This compiler does not bootstrap itself. It is written in C, which is not compatible with its implementation of C++. Therefore, there is no need to try to bootstrap it. The main incompatibility is that C-style function definitions, such as `int f (a, b) int a, b;` are beyond the grasp of GNU C++. Currently, functions **must** be declared, e.g.,
   `int f (int a, int b)`. Otherwise, the compiler may abort.
6. Install the compiler's passes and run-time support.

   Copy or link the file '`c++`' made by the compiler to the name '`/usr/local/lib/gcc-c++`'.

   Copy or link the file '`gnulib+`' made by the compiler to the name '`/usr/local/lib/gcc-gnulib+`'. This file is included automatically when GNU C++ runs the linker.

   Make the file '`/usr/local/lib/gcc-cpp+`' either a link to '`/lib/cpp+`' or a link to or copy of the file '`cpp+`' generated by '`make`'.

   Warning: the GNU CPP may not work for '`ioctl.h`', '`ttychars.h`' and other files. This cannot be fixed in the GNU CPP because the bug is in the include files: at least on some machines, they rely on behavior that is incompatible with ANSI C. This behavior consists of substituting for macro argument names when they appear inside of character constants.

   Because of this problem, you might prefer to configure GNU CC to use the system's own C preprocessor. To do so, make the file '`/usr/local/lib/gcc-cpp`' a link to '`/lib/cpp`'. This will leave C++-style comments (which begin with `//`) in the output, but the compiler will scan past them.

   Alternatively, on Sun systems and 4.3BSD at least, you can correct the include files by running the shell script '`fixincludes`'. This installs modified, corrected copies of the files '`ioctl.h`' and '`ttychars.h`' in a special directory where only GNU C++ will normally look for them.

   The file '`/usr/include/vaxuba/qvioctl.h`' used in the X window system needs a similar correction.

7. Install the compiler driver. This is the file 'g++' generated by 'make'.

If you cannot install the compiler's passes and run-time support in '/usr/local/lib', you can alternatively use the '-B' option to specify a prefix by which they may be found. The compiler concatenates the prefix with the names 'cpp+', 'c++' and 'gnulib+'. Thus, you can put the files in a directory '/usr/foo/g++' and specify '-B/usr/foo/g++/' when you run GNU C++.

If you wish to make use of the GNU C++ libraries, install the header files found in the directory 'dist-lib/incl' in a directory which cpp+ knows to search. By default, this is the directory '/usr/local/lib/g++-include'. Once these header files are installed, go to the directory 'dist-libg++/src' and just type "make". This will make the library 'libg++.a' which can then be copied to the directory /usr/lib. After copying the library, remember to run ranlib on 'libg++.a' to avoid getting a "library contents out of date" warning from the linker.

3. Trouble in Installation

Here are some of the things that have caused trouble for people installing GNU CC.

Using header files straight from '`/usr/include`' with GNU C++ will cause problems unless those header files contain fully-prototyped function declarations. A set of common header files are distributed with the GNU C++ library. These should be installed in a directory which the GNU C++ preprocessor can find them (such as '`/usr/local/lib/g++-include`').

On certain systems, defining certain environment variables such as '`CC`' can interfere with the functioning of `make`.

4. GNU C++ Header Files and Libraries

The GNU C++ compiler is a program which translates C++ code into the assembly language of a given machine. As such, it may be said that GNU C++ **implements** the C++ programming language for that machine. However, most users are accustomed to a certain amount of support beyond the bare language itself. A set of header files are provided which simplify interfacing GNU C++ code with C and UNIX routines. These header files are needed to solve the following two problems.

First, while it is optional in C to declare a function like '`printf`' before using it, in GNU C++, failure to do so results in a warning. Second, in C the declaration `int atoi()` declares that '`atoi`' is a function returning an int, while in GNU C++ that declaration would mean that function '`atoi`' *takes no arguments* and returns an int. Consequently, the following call

```
int i = atoi ("20");
```

would be tagged as an error in GNU C++ (unless the user specified the flag '`-fno-strict-prototype`' See chapter 1 [Options], page 7). The header files provided in the GNU C++ distribution provide appropriate declarations for many of the most frequently used functions. In the cases where a GNU C++ header file has the same name as a standard C header file (such as '`stdio.h`'), that header file should take precedence over the C version. This can be ensured by placing the GNU C++ header files in a directory which is always searched *before* the standard directories, such as '`/usr/include`'.

GNU C++ library provides another bridge between the bare language and its actual use. For example, the standard UNIX input/output facilities provide an untyped interface between user code and the input/output devices. Because C++ is a more strongly typed language, one would expect an input/output interface that is also strongly typed. The GNU C++ stream library provides such an interface. The stream library, and other library functions are described below.

The GNU C++ header files and public service object library are found in the directory '`dist-libg++`'. The header files are all in the sub-directory '`incl`'. The sources for the public service object library, as well as some test programs, are in the sub-directory '`src`'.

4.1 Header Files

All GNU C++ header files provided in the release conform to the following conventions. It is

recommended that these conventions be followed, especially if you wish to contribute code to the GNU C++ libraries.

- Include files that define C++ classes begin with capital letters (as do the names of the classes themselves). '`stream.h`' is uncapitalized for AT&T C++ compatibility. These files are split into two logical parts, class declarations and inlined functions. Inline functions are processed only if the files are compiled with the '`-O`' switch, which automatically defines `__OPTIMIZE__`. If the files are not compiled with '`-O`', the function versions of all inlines, kept in '`libg++.a`' are used, providing fast-compile/slow-run performance. Otherwise inlines are used extensively to achieve slow-compile/fast-run performance.
- Include files that supply function prototypes for other C functions (system calls & libraries) are all lower case.
- Note that in GNU C++, function prototypes may be declared for functions that do not actually exist. This is fine, so long as these functions are not actually used in a program.
- All include files define a preprocessor variable _X_H, where X is the name of the file, and conditionally compile only if this has not been already defined. In those cases where a file must be included more than once, one can undefine the corresponding variable.

Function prototypes in most of the following files have been created mainly by reading descriptions found in various Unix manuals. As of this writing, few have been thoroughly checked for accuracy. Most declarations appear to be correct for Vax BSD, Sun, and most SystemV based systems. Corrections are most welcome. The header files currently provided are (with a brief description of their functionality):

'`std.h`' A collection of common system calls and '`libc.a`' functions. Only those functions that can be declared without introducing new type definitions (socket structures, for example) are provided. Common `char*` functions (like `strcmp`) are among the declarations.

'`string.h`'
This file merely includes '`<std.h>`', where string function prototypes are declared. This is a workaround for the fact that system '`string.h`' and '`strings.h`' files often differ in contents.

'`math.h`' A collection of prototypes for functions usually found in libm.a, plus some `#defined` constants that appear to be consistent with those provided in the AT&T version. The value of `HUGE` should be checked before using.

'`stdio.h`' Declaration of `FILE` (`_iobuf`), lowest-common-denominator versions of common macros (like `getc`), and function prototypes for '`libc.a`' functions that operate on `FILE*`'s. The value `BUFSIZ` and the declaration of `_iobuf` should be checked before using.

'`stddef.h`'
Various useful `#define`'s and enumeration types, such as `TRUE`, `FALSE`, and `NULL`.

The type definition for pointers to libg++ error handling functions, `typedef void (*one_arg_error_handler_t)(char*);` is also given here.

'`stdarg.h`'

Definitions for vararg declarations. This is the version provided with the GNU CC distribution.

4.2 The stream class

The stream class provides an efficient, easy-to-use, and type-secure interface between GNU C++ and an underlying input/output facility, such as the one provided by UNIX. This section documents the implementation highlights of the GNU C++ stream facility. For a more complete discussion about what streams provide and how they are used, see Stroustrup's "The C++ Programming Language."

Classes `istream` and `ostream` are derived from class File. All operations are based on common C stdio library functions. They support the basic istream and ostream features described in the Stroustrup C++ book, ch. 8 (with a few minor differences) together with a few other operations based on the File class. `istream` is a non-public derived class of `File`, and only imports functions necessary for input operations. `ostreams` are similarly structured for output operations.

Many programs previously using the AT&T stream library should run with no modification. Here is a brief summary of differences between stream operations supported here versus those described by Stroustrup, ch.8:

- istreams and ostreams are derived classes of class `File` (see below) rather than new classes with `streambuf` members. Methods for opening, closing, etc., streams are a little different, although most AT&T methods are supported.
- `f >> c` for `istream f` and `char c`, behaves exactly `f.get(c)`, and does **not** skip white space.
- Similarly, `f << c` behaves like `f.put(c)`. This feature should only be used when all files are compiled with the flag `-fchar-charconst`. Otherwise, care is required when outputting single-quoted constants like `'\n'` or `'a'`. Without the `-fchar-charconst` flag, `'\n'` and all other single-quoted constants are treated as **integer** constants, so `f<<'\n'` will print `10`.
- Otherwise, the behavior of "<<" and ">>" is closer to stdio `scanf` and `printf` (upon which they are based) than are the AT&T versions. ">>" operators ignore leading whitespace before performing conversions. There is no `skipws` class variable in the stream classes to control this.
- Although istreams and ostreams may be bound to the same physical file, istreams do not possess a `tied_to` variable to control flushing of output streams tied to input streams. The

stdio I/O functions already perform this function for standard input and output, which is generally the only case in which this construct is useful.

- Streams constructed out of character buffers are not yet supported.

4.3 The File class

The `File` class supports basic IO on unix files. Operations are based on common C stdio library functions.

`File` serves as the base class for istreams, ostreams, and other derived classes. It contains the interface between the Unix stdio file library and these more structured classes. Most operations are implemented as simple calls to stdio functions. `File` class operations are also fully compatible with raw system file reads and writes (like the system `read` and `lseek` calls) when buffering is disabled (see below). The `FILE*` stdio file pointer is, however maintained as private. Classes derived from File may only use the IO operations provided by File, which encompass essentially all stdio capabilities.

Compilation of class `File` requires the existence of a suitable version of '`stdio.h`', as well as several system include files, and other include files provided with this distribution. There are also three conditional compilation flags, HAVE_VPRINTF, HAVE_SETLINEBUF, and HAVE_SETVBUF, that should be checked for correctness before compilation.

The class contains four general kinds of functions: methods for binding `File`s to physical Unix files, basic IO methods, file and buffer control methods, and methods for maintaining logical and physical file status.

4.3.1 Binding

Binding and related tasks are accomplished via `File` constructors and destructors, and member functions `open`, `close`, `remove`, `filedesc`, `name`, `setname`.

`File`s may be constructed in any of the ways supported by a version of 'open', plus a default constructor. They differ in specifying if

- a file with a given filename should be opened. The second argument refers to the IO mode (`io_readonly`, `io_readwrite`, etc.). The third represents the access mode (`a_create`, etc.).

These modes encompass those available via the system open function, but are decribed via enumeration types, rather than combinations of special flags.

- same as above, except the mode is given using the `fopen` char* string argument (`"r"`, `"w"`, `"a"`, `"r+"`, `"w+"`, `"a+"`).
- the `File` should be bound to a file associated with the given (open) file descriptor. This method should be used only if a file pointer associated with the file descriptor has not yet been obtained. The second argument specifies the io_mode, as above. This must match the actual IO mode of the file.
- the `File` should be bound to a FILE* file pointer already somehow obtained. This is mainly used to bind `Files` to the default stdin, stdout, and stderr files.
- the `File` should not yet be bound to anything. Files may be declared via this default, and then later opened via `open`.

After a successful open, the corresponding file descriptor is accessible (for use in system calls, etc.) via `filedesc()`. A `File` may be bound to different physical files at different times: each call to `open`, closes the old physical file and rebinds the `File` to a new physical file.

If a file name is provided in a constructor or open, it is maintained as class variable `nm` and is accessible via `name`. If no name is provided, then `nm` remains null, except that `Files` bound to the default files stdin, stdout, and stderr are automatically given the names `(stdin)`, `(stdout)`, `(stderr)` respectively. The function `setname` may be used to change the internal name of the `File`. This does not change the name of the physical file bound to the File.

The member function `close` closes a file. The `~File` destructor closes a file if it is open, except that stdin, stdout, and stderr are flushed but left open for the system to close on program exit since some systems may require this, and on others it does not matter. `remove` closes the file, and then deletes it if possible by calling the system function to delete the file with the name provided in the `nm` field.

4.3.2 Basic IO

- `read` and `write` perform binary IO via stdio `fread` and `fwrite`.
- `get` and `put` for chars are inline functions that invoke stdio `getc` and `putc` macros.
- `get(char* s, int maxlength, char terminator='\n')` behaves as described by Stroustrup. It reads at most maxlength characters into s, stopping when the terminator is read, and pushing the terminator back into the input stream. To accomodate different conventions about what to do about the terminator, the function `getline(char* s, int maxlength,`

`char terminator='\n'`) behaves like get, except that the terminator becomes part of the string, and is not pushed back.

- `put(const char* s)` outputs a null-terminated string via stdio `fputs`.
- `form` is a front-end for stdio `printf`, and `scan` for `scanf`. Note that the member function `form` is distinct from (and typically more useful than) the nonmember `form`.
- `unget` and `putback` are synonyms. Both call stdio `ungetc`.

4.3.3 File Control

`flush`, `seek`, `tell`, and `tell` call the corresponding stdio functions.

`setbuf` is mainly useful to turn off buffering in cases where nonsequential binary IO is being performed. `raw` is a synonym for `setbuf(_IONBF)`. After a `f.raw()`, using the stdio functions instead of the system `read`, `write`, etc., calls entails very little overhead. Moreover, these become fully compatible with intermixed system calls (e.g., `lseek(f.filedesc(), 0, 0)`). While intermixing `File` and system IO calls is not at all recommended, this technique does allow the `File` class to be used in conjuction with other functions and libraries already set up to operate on file descriptors. `setbuf` should be called at most once after a constructor or open, but before any IO.

4.3.4 File Status

File status is maintained in several ways.

A `File` may be checked for accessibility via `is_open()`, which returns true if the File is bound to a usable physical file, `readable()`, which returns true if the File can be read from (opened for reading, and not in a _fail state), or `writable()`, which returns true if the File can be written to.

`File` operations return their status via two means: failure and success are represented via the logical state. Also, the return values of invoked stdio and system functions that return useful numeric values (not just failure/success flags) are held in a class variable accessible via `iocount`. (This is useful, for example, in determining the number of items actually read by the `read` function.)

Like the AT&T i/o-stream classes, but unlike the description in the Stroustrup book, p238, `rdstate()` returns the bitwise OR of `_eof`, `_fail` and `_bad` not necessarily distinct values. The functions `eof()`, `fail()`, `bad()`, and `good()` can be used to test for each of these conditions independently.

`_fail` becomes set for any input operation that could not read in the desired data, and for other failed operations. As with all unix IO, `_eof` becomes true only when an input operations fails because of an end of file. Therefore, `_eof` is not immediately true after the last successful read of a file, but only after one final read attempt. Thus, for input operations, `_fail` and `_eof` almost always become true at the same time. `bad` is set for unbound files, and may also be set by applications in order to communicate input corruption. Conversely, `_good` is defined as 0 and is returned by `rdstate()` if all is well.

The state may be modified via `clear(flag)`, which, despite its name, sets the corresponding state_value flag. `clear()` with no arguments resets the state to _good. `failif(int cond)` sets the state to `_fail` only if `cond` is true. `failif` also invokes the function `error`. `error` in turn calls a resetable error handling function pointed to by the non-member global variable `File_error_handler` only if a system error has been generated. Since `error` cannot tell if the current system error is actually responsible for a failure, it may at times print out spurious messages. Three error handlers are provided. The default, `verbose_File_error_handler` calls the system function `perror` to print the corresponding error message on standard error, and then returns to the caller. `quiet_File_error_handler` does nothing, and simply returns. `fatal_File_error_handler` prints the error and then aborts execution. These three handlers, or any other user-defined error handlers can be selected via the non-member function `set_File_error_handler`.

All read and write operations communicate either logical or physical failure by setting the _fail flag. All further operations are blocked if the state is in a _fail or _bad condition. Programmers must explicitly use `clear()` to reset the state in order to continue and IO processing after either a logical or physical failure. C programmers who are unfamiliar with these conventions should note that, unlike the stdio library, `File` functions indicate IO success, status, or failure soley through the state, not via return values of the functions. The `void*` operator or `rdstate()` may be used to test success. In particular, according to c++ conversion rules, the `void*` coercion is automatically applied whenever the `File&` return value of any function is tested in an `if` or `while`. Thus, for example, an easy way to copy all of stdin to stdout until eof (at which point `get` fails) or some error is `char c; while(cin.get(c) && cout.put(c));`.

4.3.5 The SFile class

`SFile` (short for structure file) is provided both as a demonstration of how to build derived classes from `File`, and as a useful class for processing files containing fixed-record-length binary data. They are created with constructors with one additional argument declaring the size (in bytes, i.e, `sizeof` units) of the records. `get`, will input one record, `put` will output one, and the [] operator, as in `f[i]`, will position to the i'th record. If the file is being used mainly for random access, it is often a good idea to eliminate internal buffering via `setbuf` or `raw`. Here is an example:

```
class record
{
  friend class SFile;
  char c; int i; double d;     // or anything at all
};

void demo()
{
  record r;
  SFile recfile("mydatafile", sizeof(record), io_readwrite, a_create);
  recfile.raw();
  for (int i = 0; i < 10; ++i)  // ... write some out
  {
    r = something();
    recfile.put(&r);            // must use '&r' for proper coercion
  }
  for (i = 9; i >= 0; --i)      // now use them in reverse order
  {
    recfile[i].get(&r);
    do_something_with(r);
  }
}
```

4.3.6 The PlotFile Class

Class `PlotFile` is a simple derived class of `File` that may be used to produce files in Unix plot format. Public functions have names corresponding to those in the `plot(5)` manual entry.

4.4 The String class

The `String` class is designed to extend GNU C++ to support string processing capabilities similar to those in languages like awk. The class provides facilities that ought to be convenient and efficient enough to be useful replacements for `char*` based processing via the C string library (i.e., `strcpy`, `strcmp`, etc., in many applications.

String processing facilities usually have two major bottlenecks: storage management and copying. The `String` class avoids most such problems, at the expense of other, cheaper forms of overhead via the following strategies:

- String variables are really pointers to the actual `_Srep` representations, in a way roughly similar to that described in the Stroustrup book, p 184. This technique allows string representations to

be shared across many String variables. This can greatly reduce copying in many applications, and generally compensates for the extra level of indirection. The length of a String, its currently allocated maximum size, and its reference count are contained in the `_Srep` representation. Strings may be as long as representable by a `short int` (typically 32767 bytes), although the implementation is best tuned for manipulating Strings of length less than a hundred bytes or so.

- All dynamic allocation is controlled from within the class. Users should never need to allocate and deallocate space for Strings. Deallocation is controlled via a simple reference counting mechanism. Unfortunately, because of the differences between their allocation strategies, Strings are not well-integrated with Obstacks.
- The built-in new operator and/or the C realloc function, are used internally for allocation purposes. In order to reduce allocation and re-allocation needs, whenever a String expands as the result of some operation, it is over-allocated by about a factor of two. Thus, Strings are originally given only as much space as they need, but if there is any indication that a String might be growing, it is over-allocated.
- String processing often involve operations intermixing String variables with quoted string constants, characters, and the like. In order to avoid coercions from non-Strings into Strings in such cases, which would require otherwise useless allocation overhead, most String operations are explicitly overloaded for each supported argument type combination. In the case of infix operators, special versions are provided only for non-Strings occurring on the right-hand side of the operator, just to keep down proliferation of function definitions. The corresponding operations are performed by calling lower-level (and otherwise inaccessible) string manipulation functions with the apprpriate parameters. This strategy substitutes function call overhead for allocation overhead.
- A separate `SubString` class supports the usual substring extraction and modification operations. This is implemented in a way that user programs never directly construct or represent substrings, which are only used indirectly via String operations.
- Another separate class, `Regex` is also used indirectly via String operations in support of regular expression searching, matching, and the like. Regex capabilities are based entirely on the functions provided in GNU Emacs source file '`regex.c`'.

4.4.1 String Constructors

Strings are initialized and assigned as follows:

`String x; String y = 0;`

Set x and y to the nil string. Note that 0 (or "") may always be used to refer to the nil string.

`String x = "Hello"; String y("Hello");`
Set x and y to a copy of the string "Hello".

`String x = 'A'; String y('A');`
Set x and y to the string value "A"

`String u = x; String v(x);`
Set u and v to the same string as String x

`String u = x(1,4); String v(x(1,4));`
Set u and v to the length 4 substring of x starting at position 1.

`String x("abc", 2);`
Sets x to "ab", i.e., the first 2 characters of "abc". The second (length) argument may be greater that the length of the char* string. This form of the constructor may be used just to pre-allocate space via, for example, `String x("", 100)`, although this is rarely useful.

`String x = dec(20);`
Sets x to "20". As here, Strings may be initialized or assigned the results of any `char*` function.

There are no directly accessible forms for declaring SubString variables.

4.4.2 Regex constructors

The Regex class is based entirely on the GNU emacs regex functions. Refer to the GNU Emacs documentation for details about regular expression syntax, etc. See the internal documentation in files '`regex.h`' and '`regex.c`' for implementation details.

The declaration `Regex r("[a-zA-Z_][a-zA-Z0-9_]*");` creates a compiled regular expression suitable for use in String operations described below. (In this case, one that matches any C++ identifier). The first argument may also be a String. Be careful in distinguishing the role of backslashes in quoted GNU C++ char* constants versus those in Regexes. For example, a Regex that matches either one or more tabs or strings beginning with "ba" and ending with any number of occurrences of "na" could be declared as `Regex r = "\\(\t+\\)\\|\\(ba\\(na\\)*\\)"` Note that only one backslash is needed to signify the tab, but two are needed for the parenthesization and virgule, since the GNU C++ lexical analyzer decodes and strips backslashes before they are seen by Regex.

There are three additional optional arguments to the Regex constructor that are seldom useful:

`fast (default 0)`

`fast` may be set to true (1) if the Regex should be "fast-compiled". This causes an additional compilation step that is generally worthwhile if the Regex will be used many times.

`bufsize (default 40)`

This is an estimate of the size of the internal compiled expression. Set it to a larger value if you know that the expression will require a lot of space. If you do not know, do not worry: realloc is used if necessary.

`transtable (default none == 0)`

The address of a byte translation table (a char[256]) that translates each character before matching.

As a convenience, several Regexes are predefined and usable in any program. Here are their declarations from '`String.h`'.

```
extern Regex RXwhite;          // = "[ \n\t]+"
extern Regex RXint;            // = "-?[0-9]+"
extern Regex RXdouble;         // = "-?\\(\\([0-9]+\\.[0-9]*\\)\\|
                               //    \\([0-9]+\\)\\|\\(\\.[0-9]+\\)\\)
                               //    \\([eE][---+]?[0-9]+\\)?"
extern Regex RXalpha;          // = "[A-Za-z]+"
extern Regex RXlowercase;      // = "[a-z]+"
extern Regex RXuppercase;      // = "[A-Z]+"
extern Regex RXalphanum;       // = "[0-9A-Za-z]+"
extern Regex RXidentifier;     // = "[A-Za-z_][A-Za-z0-9_]*"
```

4.4.3 examples

Most `String` class capabilities are best shown via example. The examples below use the following declarations.

```
String x = "Hello";
String y = "world";
String n = "123";
String z;
char*  s = ",";
String lft, mid, rgt;
Regex  r = "e[a-z]*o";
Regex  r2("/[a-z]*/");
char   c;
int    i, pos, len;
double f;
```

```
String words[10];
words[0] = "a";
words[1] = "b";
words[2] = "c";
```

4.4.4 Matching

The usual lexigraphic relational operators (`==`, `!=`, `<`, `<=`, `>`, `>=`) are defined.

Other matching operations are based on some form of the `index` function. As seen in the following examples, the second optional `startpos` argument to `index` and all other operations involving search specifies the starting position of the search: If non-negative, it results in a left-to-right search starting at position `startpos`, and if negative, a right-to-left search starting at position `x.length() - startpos`. In all cases, the index returned is that of the beginning of the match, or -1 if there is no match.

`x.index("lo")`
: returns the zero-based index of the leftmost occurence of substring "lo" (3, in this case). The argument may be a String, SubString, char, char*, or Regex.

`x.index("l", 2)`
: returns the index of the first of the leftmost occurence of "l" found starting the search at position x[2], or 2 in this case.

`x.index("l", -1)`
: returns the index of the rightmost occurence of "l", or 3 here.

`x.index("l", -3)`
: returns the index of the righmost occurence of "l" found by starting the search at the 3rd to the last position of x, returning 2 in this case.

`pos = r.search("leo", 3, len, 0)`
: returns the index of r in the `char*` string of length 3, starting at position 0, also placing the length of the match in reference parameter len.

`x.contains("He")`
: returns true if the String x contains the substring "He". The argument may be a String, SubString, char, or char*, or Regex.

`x.contains(RXwhite);`
: returns true if x contains any whitespace (space, tab, or newline). Recall that `RXwhite` is a global whitespace Regex.

`x.contains(r)`
: returns true if x contains any instance of the Regex r.

`x.matches(r)`

returns true if String x as a whole matches Regex r.

4.4.5 Substring extraction

Substrings may be extracted via the `at`, `before` and `after` functions. These behave as either lvalues or rvalues.

`z = x.at(2, 3)`

sets String z to be equal to the length 3 substring of String x starting at zero-based position 2, setting z to "llo" in this case. A nil String is returned if the arguments don't make sense.

`x.at(2, 2) = "r"`

Sets what was in positions 2 to 3 of x to "r", setting x to "Hero" in this case. As indicated here, SubString assignments may be of different lengths.

`x.at("He") = "je";`

x("He") is the substring of x that matches the first occurence of it's argument. The substitution sets x to "jello". If "He" did not occur, the substring would be nil, and the assignment would have no effect.

`x.at("l", -1) = "i";`

replaces the rightmost occurence of "l" with "i", setting x to "Helio".

`z = x.at(r)`

sets String z to the match in x of Regex r, or "ello" in this case. A nil String is returned if there is no match.

`z = x.before("o")`

sets z to the part of x to the left of the first occurrence of "o", or "Hell" in this case. The argument may also be a String, SubString, or Regex.

`x.before("ll") = "Bri";`

sets the part of x to the left of "ll" to "Bri", setting x to "Brillo".

`z = x.before(2)`

sets z to the part of x to the left of x[2], or "He" in this case.

`z = x.after("Hel")`

sets z to the part of x to the right of "Hel", or "lo" in this case.

`x.after("Hel") = "p";`

sets x to "Help";

`z = x.after(3)`

sets z to the part of x to the right of x[3] or "o" in this case.

`z = " ab c"; z = z.after(RXwhite)`
sets z to the part of its old string to the right of the first group of whitespace, setting z to "ab c"; Use gsub(below) to strip out multiple occurences of whitespace or any pattern.

4.4.6 Concatenation

`z = x + s + ' ' + y.at("w") + y.after("w") + ".";`
sets z to "Hello, world."

`x += y;` sets x to "Helloworld"

`z = replicate(x, 3);`
sets z to "HelloHelloHello".

`z = join(words, 3, "/")`
sets z to the concatenation of the first 3 Strings in String array words, each separated by "/", setting z to "a/b/c" in this case. The last argument may be any of the usual, including "" or 0, for no separation.

4.4.7 String manipulation

`z = "left/middle/right"; decompose(z, lft, mid, rgt, r2);`
sets lft to the part of z to the left of the match via Regex r2, mid to the match, and rgt to the part to the right of the match, setting lft = "left", mid = "/middle/", and rgt to "right" in this case. The last argument may be any of the usual. If there is no match, lft, mid, and rgt remain unchanged, and decompose returns 0.

`z = "this string has five words"; i = split(z, words, 10, RXwhite);`
sets up to 10 elements of String array words to the parts of z separated by whitespace, and returns the number of parts actually encountered (5 in this case). Here, words[0] = "this", words[1] = "string", etc. The last argument may be any of the usual. If there is no match, all of z ends up in words[0]. The words array is **not** dynamically created by split.

`x.gsub("l","ll")`
substitutes all original occurrences of "l" with "ll", setting x to "Hellllo". The first argument may be any of the usual, including Regex. If the second argument is "" or 0, all occurences are deleted.

`z = x + y; z.del("loworl");`
deletes the leftmost occurence of "loworl" in z, setting z to "Held".

`z = reverse(x)`
: sets z to the reverse of x, or "olleH".

`z = upcase(x)`
: sets z to x, with all letters set to uppercase, setting z to "HELLO"

`z = downcase(x)`
: sets z to x, with all letters set to lowercase, setting z to "HELLO"

4.4.8 Reading and writing

`cout << x`
: writes out x. cout.put(x) has the same effect.

`cout << x(2, 3)`
: writes out the substring "llo".

`cin >> x` reads a whitespace-bounded string into x.

`cin.get(x, 100)`
: reads up to 100 characters into x, stopping at a newline.

`cin.getline(x, 100)`
: reads up to 100 characters into x, stopping at, but including, a newline.

4.4.9 Conversion

`x.length()`
: returns the length of String x (5, in this case).

`s = (char*)x`
: can be used to extract the `char*` char array. This coercion is useful for sending a String as an argument to any function expecting a `const char*` argument (like `atoi`, and `File::open`). This operator must be used with care. Strings should not be **modified** by nonmember functions. Doing so may corrupt their representation. The conversion is defined to return a const value so that GNU C++ will produce warning and/or error messages if changes are attempted. In cases where the String must be modified via a function taking a `char*` argument, the `make_unique` member function may be employed. This forces x to point to an unshared string representation. For example, if for some reason, a String needed to be changed via `strcpy`, `x.make_unique(); strcpy(x, "Hi");` would generate a compiler warning, but would work corectly so long as x already possessed sufficient space. Again, this is not a recommended practice.

`c = x[i]` returns the **value** of the i'th character of x. The value of i is not checked against the bounds of the string. (All this ensures that using elements of x[i] for, e.g., computing a hash function is as efficient as using raw char* indexing.) Since the value, and not the reference is returned, `x[i] = 'a';` does not work. This sort of operation can be performed via the SubString operators as in `x.at(i, 1) = "a";`.

4.5 The Integer class.

The `Integer` class provides multiple precision integer arithmatic facilties. `Integers` are represented using a reference-counting dynamic allocation technique almost exactly the same as used in class `String`.

`Integers` may be up to `b * ((1 << b) - 1)` bits long, where `b` is the number of bits per short (typically 1048560 bits when `b = 16`). The implementation file '`Integer.cc`' contains some machine-dependent constants that should be checked for accuracy before compilation. The implementation assumes that a `long` is at least twice as long as a `short`. This assumption hides beneath almost all primitive operations, and would be very difficult to change. It also relies on correct behaviour of *unsigned* arithmetic operations.

Some of the arithmetic algorithms are loosely based on those provided in the MIT Scheme '`bignum.c`' release, which is Copyright (c) 1987 Massachusetts Institute of Technology. Their use here falls within the provisions described in the Scheme release.

Integers may be declared and intitialized via

`Integer x;`
: Declares an unitialized Integer.

`Integer x = 2; Integer y(2);`
: Set x and y to the Integer value 2;

`Integer u(x); Integer v = x;`
: Set u and v to the same value as x.

`Integers` may be coerced back into longs via the `long` coercion operator. If the Integer cannot fit into a long, this returns MINLONG or MAXLONG (depending on the sign) where MINLONG is the most negative, and MAXLONG is the most positive representable long. The member function `fits_in_long()` may be used to test this.

All of the usual arithmetic operators are provided (`+`, `-`, `*`, `/`, `%`, `+=`, `++`, `-=`, `--`, `*=`, `/=`, `%=`, `==`, `!=`, `<`, `<=`, `>`, `>=`). All operators support special versions for mixed arguments of Integers and regular C++ longs in order to avoid useless coercions, as well as to allow automatic promotion of shorts and ints to longs, so that they may be applied without additional Integer coercion operators. The only operators that behave differently than the corresponding int or long operators are `++` and `--`. Because C++ does not distinguish prefix from postfix application, these are declared as `void` operators, so that no confusion can result from applying them as postfix. Thus, for Integers x and y, `++x; y = x;` is correct, but `y = ++x;` and `y = x++;` are not.

Bitwise operators (`~`, `&`, `|`, `^`, `<<`, `>>`, `&=`, `|=`, `^=`, `<<=`, `>>=`) are also provided. However, these operate on sign-magnitude, rather than two's complement representations. The sign of the result is arbitrarily taken as the sign of the first argument. For example, `Integer(-3) & Integer(5)` returns `Integer(-1)`, not -3, as it would using two's complement. Also, `~`, the complement operator, complements bits up to the next `short` boundary of the representation. While arbitrary, this effect may be useful when combined with other bitwise operations.

Several other common integer functions are available. For compatibility, many corresponding `long` and mixed argument functions are also implemented.

`void divide(x, y, q, r);`
: Sets q to the quotient and r to the remainder of x and y. (q and r are passed and returned by reference)

`Integer pow(x, p)`
: returns x raised to the power p.

`Integer gcd(x, y)`
: returns the greatest common divisor of x and y.

`Integer abs(x);`
: returns the absolute value of x.

`Integer sqr(x)`
: returns x * x;

`Integer sqrt(x)`
: returns the floor of the square root of x.

`Integer rnd(x)`
: returns a random number between 0 and x-1, or between x+1 and 0 if x is negative. This function uses the standard libc rand().

`long lg(x);`
: returns the floor of the base 2 logarithm of abs(x)

`int sign(x)`
: returns -1 if x is negative, 0 if zero, else +1. Using `if (sign(x) == 0)` is a generally

faster method of testing for zero than using relational operators.

`int even(x)`

returns true if x is an even number

`int odd(x)`

returns true if x is an odd number.

`void bitset(Integer& x, long b)`

sets the b'th bit (counting right-to-left from zero) of x to 1.

`void bitclear(Integer& x, long b)`

sets the b'th bit of x to 0.

`int bittest(Integer x, long b)`

returns true if the b'th bit of x is 1.

`Integer atoI("1234567");`

converts the char* string into its Integer form.

`char* Itoa(x);`

returns a (static) pointer to the ascii string value of x. The static buffer is of fixed size (BUFSIZ, typically 1024). Conversion of very large integers (>= pow(10, BUFSIZ)) causes an exception.

Several other member functions are available that were designed mainly for internal use, but are conceivably useful in other contexts as well.

`int x.cmp(Integer y)`

returns a negative number if x<y, zero if x==y, or positive if x>y.

`int x.ucmp(Integer y)`

like cmp, but performs unsigned comparison.

`void x.setlength(long len)`

pre-allocates len shorts for x.

`void x.make_unique()`

forces x to have a unique (unshared) Irep pointer.

`void x.error(char* msg)`

Calls `*Integer_error_handler`. This is called internally when division by zero and similar exceptions occur. The default error handler prints the error message and aborts execution.

4.6 Obstacks

The `Obstack` class is a simple rewrite of the C obstack macros and functions provided in the

GNU CC compiler source distribution.

Obstacks provide a simple method of creating and maintaining a string table, optimized for the very frequent task of building strings character-by-character, and sometimes keeping them, and sometimes not. They seem especially useful in any parsing application. One of the test files demonstrates usage.

A brief summary:

`grow` places something on the obstack without committing to wrap it up as a single entity yet.

`finish` wraps up a constructed object as a single entity, and returns the pointer to its start address.

`copy` places things on the obstack, and *does* wrap them up. `copy` is always equivalent to first grow, then finish.

`free` deletes something, and anything else put on the obstack since its creation.

The other functions are hardly ever needed:

`blank` is like grow, except it just grows the space by size units without placing anything into this space

`alloc` corresponds in the same way to `copy`.

`chunk_size, base, etc.`
just return class variables.

`grow_fast`
places a character on the obstack without checking if there is enough room.

Here is a lightly edited version of the original C documentation:

These functions operate a stack of objects. Each object starts life small, and may grow to maturity. (Consider building a word syllable by syllable.) An object can move while it is growing. Once it has been "finished" it never changes address again. So the "top of the stack" is typically an immature growing object, while the rest of the stack is of mature, fixed size and fixed address objects.

These routines grab large chunks of memory, using the GNU C++ `new` operator. On occasion, they free chunks, via `delete`.

Each independent stack is represented by a Obstack.

One motivation for this package is the problem of growing char strings in symbol tables. Unless you are a "facist pig with a read-only mind" [Gosper's immortal quote from HAKMEM item 154, out of context] you would not like to put any arbitrary upper limit on the length of your symbols.

In practice this often means you will build many short symbols and a few long symbols. At the time you are reading a symbol you don't know how long it is. One traditional method is to read a symbol into a buffer, `realloc()`ating the buffer every time you try to read a symbol that is longer than the buffer. This is beaut, but you still will want to copy the symbol from the buffer to a more permanent symbol-table entry say about half the time.

With obstacks, you can work differently. Use one obstack for all symbol names. As you read a symbol, grow the name in the obstack gradually. When the name is complete, finalize it. Then, if the symbol exists already, free the newly read name.

The way we do this is to take a large chunk, allocating memory from low addresses. When you want to build a symbol in the chunk you just add chars above the current "high water mark" in the chunk. When you have finished adding chars, because you got to the end of the symbol, you know how long the chars are, and you can create a new object. Mostly the chars will not burst over the highest address of the chunk, because you would typically expect a chunk to be (say) 100 times as long as an average object.

In case that isn't clear, when we have enough chars to make up the object, *they are already contiguous in the chunk* (guaranteed) so we just point to it where it lies. No moving of chars is needed and this is the second win: potentially long strings need never be explicitly shuffled. Once an object is formed, it does not change its address during its lifetime.

When the chars burst over a chunk boundary, we allocate a larger chunk, and then copy the partly formed object from the end of the old chunk to the beggining of the new larger chunk. We then carry on accreting characters to the end of the object as we normaly would.

A special version of grow is provided to add a single char at a time to a growing object.

Summary:

- We allocate large chunks.
- We carve out one object at a time from the current chunk.

- Once carved, an object never moves.
- We are free to append data of any size to the currently growing object.
- Exactly one object is growing in an obstack at any one time.
- You can run one obstack per control block.
- You may have as many control blocks as you dare.
- Because of the way we do it, you can 'unwind' a obstack back to a previous state. (You may remove objects much as you would with a stack.)

The obstack data structure is used in many places in the GNU C++ compiler.

Differences from the the GNU C version

1. The obvious differences stemming from the use of classes and inline functions instead of structs and macros. The C `init` and `begin` macros are replaced by constructors.
2. Overloaded function names are used for grow (and others), rather than the C `grow`, `grow0`, etc.
3. All dynamic allocation uses the the built-in `new` operator. This restricts flexibility by a little, but maintains compatibility with usual C++ conventions. Also, users can always redefine `new` and `delete` for this class.
4. There are now two versions of finish:
   1. finish() behaves like the C version.
   2. finish(char terminator) adds `terminator`, and then calls `finish()`. This enables the normal invocation of `finish(0)` to wrap up a string being grown character-by-character.
5. There are special versions of grow(const char* s) and copy(const char* s) that add the null-terminated string `s` after computing its length.

5. Incompatibilities of GNU C++

There are several noteworthy incompatibilities between GNU C++ and most versions of C++ and/or C.

Ultimately our intention is that the '`-traditional`' option will eliminate all the incompatibilities that can be eliminated by telling GNU C++ to behave like the other C++/C compiler combinations.

- GNU C++ normally makes string constants read-only. If several identical-looking string constants are used, GNU C++ stores only one copy of the string.

  One consequence is that you cannot call `mktemp` with a string constant argument. The function `mktemp` always alters the string its argument points to.

  Another consequence is that `sscanf` does not work on some systems when passed a string constant as its format control string. This is because `sscanf` incorrectly tries to write into the string constant.

  The best solution to these problems is to change the program to use `char`-array variables with initialization strings for these purposes instead of string constants. But if this is not possible, you can use the '`-fwritable-strings`' flag, which directs GNU CC to handle string constants the same way most C compilers do.
- GNU C++ does not substitute macro arguments when they appear inside of string constants. For example, the following macro in GNU C++

  ```
  #define foo(a) "a"
  ```

  will produce output '`"a"`' regardless of what the argument a is.

  The '`-traditional`' option directs GNU CC to handle such cases (among others) in the old-fashioned (non-ANSI) fashion.
- When you use `setjmp` and `longjmp`, the only automatic variables guaranteed to remain valid are those declared `volatile`. This is a consequence of automatic register allocation. Consider this function:

  ```
  jmp_buf j;

  foo ()
  {
    int a, b;

    a = fun1 ();
    if (setjmp (j))
      return a;

    a = fun2 ();
  ```

```
    /* longjmp (j) may be occur in fun3. */
    return a + fun3 ();
}
```

Here `a` may or may not be restored to its first value when the `longjmp` occurs. If `a` is allocated in a register, then its first value is restored; otherwise, it keeps the last value stored in it.

If you use the '`-W`' option with the '`-O`' option, you will get a warning when GNU C++ thinks such a problem might be possible.

- Declarations of external variables and functions within a block apply only to the block containing the declaration. In other words, they have the same scope as any other declaration in the same place.

  In some other C++/C compiler systems, a `extern` declaration affects all the rest of the file even if it happens within a block.

  The '`-traditional`' option directs GNU C++ to treat all `extern` declarations as global, like traditional compilers.

- In traditional C, you can combine `long`, etc., with a typedef name, as shown here:

```
typedef int foo;
typedef long foo bar;
```

  In ANSI C, this is not allowed: `long` and other type modifiers require an explicit `int`. Because this criterion is expressed by Bison grammar rules rather than C code, the '`-traditional`' flag cannot alter it.

- When compiling functions that return structures or unions, GNU C++ output code uses a method different from that used on most versions of Unix. As a result, code compiled with GNU C++ cannot call a structure-returning function compiled with PCC, and vice versa.

  The method used by GNU C++ is as follows: a structure or union which is 1, 2, 4 or 8 bytes long is returned like a scalar. A structure or union with any other size is stored into an address supplied by the caller in a special, fixed register. (Structures which have constructors are passed by value via the special register, regardless of their size.)

  PCC usually handles all sizes of structures and unions by returning the address of a block of static storage containing the value. This method is not used in GNU C++ because it is slower and nonreentrant.

  On systems where PCC works this way, you may be able to make GNU C++-compiled code call such functions that were compiled with PCC by declaring them to return a pointer to the structure or union instead of the structure or union itself. For example, instead of this:

```
struct foo nextfoo ();
```

  write this:

```
struct foo *nextfoo ();
#define nextfoo *nextfoo
```

  (Note that this assumes you are using the GNU preprocessor, so that the ANSI antirecursion rules for macro expansions are effective.)

- Member functions which are declared in the scope of a class declaration are implicitly declared inline in C++. In GNU C++, the inline declaration must be explicitly specified in order to take effect. This makes it possible to keep functions from being integrated without changing a great deal of code (there is no `noinline` specifier). If you want functions to be inlined as much as possible, use the `-finline-functions` flag.

  I am considering reversing the polarity of this option, and providing a -fno-default-inline-functions flag. I am interested in hearing what people think about this.
- The naming convention of GNU C++ and AT&T C++ for overloaded functions (and member functions) are incompatible. You cannot use AT&T C++ libraries with GNU C++. This should not be a problem for long: we have already started receiving contributions in this area, and should be able to provide the most used libraries very soon.
- The ANSI draft stipulates an interpretation for items declared `const` which is incompatible with C++. GNU C++ makes an attempt to support both interpretations, using a flag to select between them. Currently, GNU C++ does not fully support the C++ interpretation of `const`. To have the effect of declaring a variable value constant, you must specify `static` for that variable. Otherwise, it is unclear whether storage should be reserved for that variable, as well as how that variable should be initialized. It is hoped that these issues will be resolved in a satisfactory way in the future.
- The syntactic form `xyzzy lose(frob);` where `xyzzy` is an aggregate type and `frob` is an object of that type, is a declaration of a new aggregate object `lose`. Under AT&T C++, this interpretation could lead to the calling of a constructor if one exists for that argument list, or it could have the equivalent meaning of the form `xyzzy lose = frob`. This duality is non-intuitive, and although implemented, is discouraged.
- The design of the C++ programming language did not take into account the usefulness of being able to specify that language using an LALR(1) grammar. As a result, in order to correctly parse it, one needs a look-ahead lexical analyzer (with infinite lookahead), and a recursive descent parser, guided by some good heuristics. This approach was not taken in GNU C++, because it is considered archaic, notoriously difficult to extend syntactically, and generally offensive. GNU C++ uses an LALR(1) grammar so that users can easily understand, and readily modify the compiler to suit their needs. Free software is useless if it becomes captive to an inaccessible or undesirable technology. However, in providing such a grammar, some syntactic forms were lost, most notably old-style C function declarations and occasionally function parameters which are declared longhand to be pointers to functions are not recognized properly. The first problem is solved by converting old-style C code to the ANSI-standard function prototype form. The second problem can always be solved by using a `typedef` for the pointer to function, and working from there. Another hack which can be used, if the parameter can legitimately be declared with a storage class (such as 'register', or 'auto') is to make that storage class explicit: `int f (register int (*pf)(int,int)) {...}`.

In addition to the syntactic problems mentioned above, the C++ language suffers from another

deficiency due to its layering on top of C. When a function is declared overloaded, some or all variants of that function must be renamed in order that they not conflict. The way that the AT&T C++ compiler accomplishes this is by not renaming the first function, but overloading all subsequent functions. As a result, when the first function declared in one compilation module is not the first such one declared in all other compilation modules, the AT&T C++ compiler may generate incorrect calls, or function definitions may conflict at link time. The GNU C++ compiler avoids this problem by renaming all overloaded functions. However, this has the negative feature that when a function is declared, and then subsequently declared as a 'friend' (or is otherwise overloaded), the function will end up being inconsistently declared. There is really no way around this problem except to be very careful when using overloaded functions. In GNU C++, all functions which are intended to be friend functions should be declared overloaded before they are declared friends.

At first blush, this restriction would make it appear that it was not possible to overload functions which are in C libraries. This is not the case. Suppose one wishes to use the library function '`random`', which takes no arguments, and then define another '`random`' function which does. The GNU C++ compiler would rename both of these functions, *but the name internal to the compiler need not be the actually assembly name written out.* This can be solved use a feature of GNU C++ (See section 6.15 [Asm Labels], page 62) as follows:

```
overload random;

extern int random (void) asm ("_random");
int random (int);
```

The following macro might be used for this purpose:

```
#define C_EXTERN(RETVAL, FNAME, ARGTYPES)  \
  extern RETVAL FNAME ARGTYPES asm ("_" # FNAME)

C_EXTERN (int, random, (int));
```

6. GNU Extensions to the C++ Language

GNU C++ provides several language features not found in ANSI standard C. (The '`-pedantic`' switch directs GNU C++ to print a warning message if any of these features is used.) To check for the availability of these features, check for a predefined macro `__GNUC__`, which is always defined under GNU C++.

6.1 Statements and Declarations inside of Expressions

A compound statement in parentheses may appear inside an expression in GNU C. This allows you to declare variables within an expression. For example:

```
({ int y = foo (); int z;
   if (y > 0) z = y;
   else z = - y;
   z; })
```

is a valid (though slightly more complex than necessary) expression for the absolute value of `foo ()`.

This feature is especially useful in making macro definitions "safe" (so that they evaluate each operand exactly once). For example, the "maximum" function is commonly defined as a macro in standard C as follows:

```
#define max(a,b) ((a) > (b) ? (a) : (b))
```

But this definition computes either *a* or *b* twice, with bad results if the operand has side effects. In GNU C, if you know the type of the operands (here let's assume `int`), you can define the macro safely as follows:

```
#define maxint(a,b) \
  ({int _a = (a), _b = (b); _a > _b ? _a : _b; })
```

Embedded statements are not allowed in constant expressions, such as the value of an enumeration constant, the width of a bit field, or the initial value of a static variable.

If you don't know the type of the operand, you can still do this, but you must use `typeof` (see

section 6.3 [Typeof], page 52) or type naming (see section 6.2 [Naming Types], page 52).

6.2 Naming an Expression's Type

You can give a name to the type of an expression using a `typedef` declaration with an initializer. Here is how to define *name* as a type name for the type of *exp*:

```
typedef name = exp;
```

This is useful in conjunction with the statements-within-expressions feature. Here is how the two together can be used to define a safe "maximum" macro that operates on any arithmetic type:

```
#define max(a,b) \
  ({typedef _ta = (a), _tb = (b);  \
    _ta _a = (a); _tb _b = (b);     \
    _a > _b ? _a : _b; })
```

The reason for using names that start with underscores for the local variables is to avoid conflicts with variable names that occur within the expressions that are substituted for `a` and `b`. Eventually we hope to design a new form of declaration syntax that allows you to declare variables whose scopes start only after their initializers; this will be a more reliable way to prevent such conflicts.

6.3 Referring to a Type with `typeof`

Another way to refer to the type of an expression is with `typeof`. The syntax of using of this keyword looks like `sizeof`, but the construct acts semantically like a type name defined with `typedef`.

There are two ways of writing the argument to `typeof`: with an expression or with a type. Here is an example with an expression:

```
typeof (x[0](1))
```

This assumes that `x` is an array of functions; the type described is that of the values of the functions.

Here is an example with a typename as the argument:

```
typeof (int *)
```

Here the type described is that of pointers to `int`.

A `typeof`-construct can be used anywhere a typedef name could be used. For example, you can use it in a declaration, in a cast, or inside of `sizeof` or `typeof`.

- This declares `y` with the type of what `x` points to.

  ```
  typeof (*x) y;
  ```

- This declares `y` as an array of such values.

  ```
  typeof (*x) y[4];
  ```

- This declares `y` as an array of pointers to characters:

  ```
  typeof (typeof (char *)[4]) y;
  ```

  It is equivalent to the following traditional C declaration:

  ```
  char *y[4];
  ```

  To see the meaning of the declaration using `typeof`, and why it might be a useful way to write, let's rewrite it with these macros:

  ```
  #define pointer(T)  typeof(T *)
  #define array(T, N) typeof(T [N])
  ```

  Now the declaration can be rewritten this way:

  ```
  array (pointer (char), 4) y;
  ```

  Thus, '`array (pointer (char), 4)`' is the type of arrays of 4 pointers to `char`.

6.4 Generalized Lvalues

Compound expressions, conditional expressions and casts are allowed as lvalues provided their operands are lvalues. This means that you can take their addresses or store values into them.

For example, a compound expression can be assigned, provided the last expression in the sequence is an lvalue. These two expressions are equivalent:

```
(a, b) += 5
a, (b += 5)
```

Similarly, the address of the compound expression can be taken. These two expressions are equivalent:

```
&(a, b)
a, &b
```

A conditional expression is a valid lvalue if its type is not void and the true and false branches are both valid lvalues. For example, these two expressions are equivalent:

```
(a ? b : c) = 5
(a ? b = 5 : (c = 5))
```

A cast is a valid lvalue if its operand is valid. Taking the address of the cast is the same as taking the address without a cast, except for the type of the result. For example, these two expressions are equivalent (but the second may be valid when the type of 'a' does not permit a cast to 'int *').

```
&(int *)a
(int **)&a
```

A simple assignment whose left-hand side is a cast works by converting the right-hand side first to the specified type, then to the type of the inner left-hand side expression. After this is stored, the value is converter back to the specified type to become the value of the assignment. Thus, if 'a' has type 'char *', the following two expressions are equivalent:

```
(int)a = 5
(int)(a = (char *)5)
```

An assignment-with-arithmetic operation such as '+=' applied to a cast performs the arithmetic using the type resulting from the cast, and then continues as in the previous case. Therefore, these two expressions are equivalent:

```
(int)a += 5
(int)(a = (char *) ((int)a + 5))
```

6.5 Conditional Expressions with Omitted Middle-Operands

The middle operand in a conditional expression may be omitted. Then if the first operand is nonzero, its value is the value of the conditional expression.

Therefore, the expression

```
x ? : y
```

has the value of x if that is nonzero; otherwise, the value of y.

This example is perfectly equivalent to

```
x ? x : y
```

In this simple case, the ability to omit the middle operand is not especially useful. When it becomes useful is when the first operand does, or may (if it is a macro argument), contain a side effect. Then repeating the operand in the middle would perform the side effect twice. Omitting the middle operand uses the value already computed without the undesirable effects of recomputing it.

6.6 Arrays of Length Zero

Zero-length arrays are allowed in GNU C++. They are very useful as the last element of a structure which is really a header for a variable-length object:

```
struct line {
  int length;
  char contents[0];
};

{
  struct line *thisline
    = (struct line *) malloc (sizeof (struct line) + this_length);
  thisline->length = thislength;
}
```

In standard C, you would have to give `contents` a length of 1, which means either you waste space or complicate the argument to `malloc`.

6.7 Arrays of Variable Length

Variable-length automatic arrays are allowed in GNU C. These arrays are declared like any other automatic arrays, but with a length that is not a constant expression. The storage is allocated at that time and deallocated when the brace-level is exited. For example:

```
FILE *concat_fopen (char *s1, char *s2, char *mode)
{
  char str[strlen (s1) + strlen (s2) + 1];
  strcpy (str, s1);
  strcat (str, s2);
  return fopen (str, mode);
}
```

You can also define structure types containing variable-length arrays, and use them even for arguments or function values, as shown here:

```
int foo;

struct entry
{
  char data[foo];
};

struct entry
tester (struct entry arg)
{
  struct entry new;
  int i;
  for (i = 0; i < foo; i++)
    new.data[i] = arg.data[i] + 1;
  return new;
}
```

(Eventually there will be a way to say that the size of the array is another member of the same structure.)

The length of an array computed on entry to the brace-level where the array is declared and is remembered for the scope of the array in case you access it with `sizeof`.

Jumping or breaking out of the scope of the array name will also deallocate the storage. Jumping into the scope is not allowed; you will get an error message for it.

You can use the function `alloca` to get an effect much like variable-length arrays. The function `alloca` is available in many other C implementations (but not in all). On the other hand, variable-length arrays are more elegant.

There are other differences between these two methods. Space allocated with `alloca` exists until the containing *function* returns. The space for a variable-length array is deallocated as soon as the

array name's scope ends. (If you use both variable-length arrays and `alloca` in the same function, deallocation of a variable-length array will also deallocate anything more recently allocated with `alloca`.)

6.8 Non-Lvalue Arrays May Have Subscripts

Subscripting is allowed on arrays that are not lvalues, even though the unary '`&`' operator is not. For example, this is valid in GNU C though not valid in other C dialects:

```
struct foo {int a[4];};

struct foo f();

bar (int index)
{
  return f().a[index];
}
```

6.9 Arithmetic on `void`-Pointers and Function Pointers

In GNU C, addition and subtraction operations are supported on pointers to `void` and on pointers to functions. This is done by treating the size of a `void` or of a function as 1.

A consequence of this is that `sizeof` is also allowed on `void` and on function types, and returns 1.

6.10 Non-Constant Initializers

The elements of an aggregate initializer are not required to be constant expressions in GNU C. Here is an example of an initializer with run-time varying elements:

```
foo (float f, float g)
{
  float beat_freqs[2] = { f-g, f+g };
  ...
}
```

6.11 Constructor Expressions

GNU C supports constructor expressions. A constructor looks like a cast containing an initializer. Its value is an object of the type specified in the cast, containing the elements specified in the initializer. The type must be a structure, union or array type.

Assume that `struct foo` and `structure` are declared as shown:

```
struct foo {int a; char b[2];} structure;
```

Here is an example of constructing a '`struct foo`' with a constructor:

```
structure = ((struct foo) {x + y, 'a', 0});
```

This is equivalent to writing the following:

```
{
  struct foo temp = {x + y, 'a', 0};
  structure = temp;
}
```

You can also construct an array. If all the elements of the constructed array are (made up of) simple constant expressions, suitable for use in initializers, then the constructor is an lvalue and can be coerced to a pointer to its first element, as shown here:

```
char **foo = (char *[]) { "x", "y", "z" };
```

Array constructors whose elements are not simple constants are not very useful, because the constructor is not an lvalue. There are only two valid ways to use it: to subscript it, or initialize an array variable with it. The former is probably slower than a `switch` statement, while the latter does the same thing an ordinary C initializer would do.

```
output = ((int[]) { 2, x, 28 }) [input];
```

6.12 Dollar Signs in Identifier Names

In GNU C, you may use dollar signs in identifier names. This is because many traditional C

implementations allow such identifiers.

6.13 Inquiring about the Alignment of a Type or Variable

The keyword `__alignof` allows you to inquire about how an object is aligned, or the minimum alignment usually required by a type. Its syntax is just like `sizeof`.

For example, if the target machine requires a `double` value to be aligned on an 8-byte boundary, then `__alignof (double)` is 8. This is true on many RISC machines. On more traditional machine designs, `__alignof (double)` is 4 or even 2.

Some machines never actually require alignment; they allow reference to any data type even at an odd addresses. For these machines, `__alignof` reports the *recommended* alignment of a type.

When the operand of `__alignof` is an lvalue rather than a type, the value is the largest alignment that the lvalue is known to have. It may have this alignment as a result of its data type, or because it is part of a structure and inherits alignment from that structure. For example, after this declaration:

```
struct foo { int x; char y; } foo1;
```

the value of `__alignof (foo1.y)` is probably 2 or 4, the same as `__alignof (int)`, even though the data type of `foo1.y` does not itself demand any alignment.

6.14 Assembler Instructions with C Expression Operands

In an assembler instruction using `asm`, you can now specify the operands of the instruction using C expressions. This means no more guessing which registers or memory locations will contain the data you want to use.

You must specify an assembler instruction template much like what appears in a machine description, plus an operand constraint string for each operand.

For example, here is how to use the 68881's `fsinx` instruction:

```
asm ("fsinx %1,%0" : "=f" (result) : "f" (angle));
```

Here `angle` is the C expression for the input operand while `result` is that of the output operand. Each has `'"f"'` as its operand constraint, saying that a floating-point register is required. The constraints use the same language used in the machine description (See the `Constraints` section in the "Internals of GNU CC" document).

Each operand is described by an operand-constraint string followed by the C expression in parentheses. A colon separates the assembler template from the first output operand, and another separates the last output operand from the first input, if any. Commas separate output operands and separate inputs. The number of operands is limited to the maximum number of operands in any instruction pattern in the machine description.

Output operand expressions must be lvalues, and there must be at least one of them. The compiler can check this. The input operands need not be lvalues, and there need not be any. The compiler cannot check whether the operands have data types that are reasonable for the instruction being executed.

The output operands must be write-only; GNU C++ will assume that the values in these operands before the instruction are dead and need not be generated. For an operand that is read-write, you must logically split its function into two separate operands, one input operand and one write-only output operand. The connection between them is expressed by constraints which say they need to be in the same location when the instruction executes. You can use the same C++ expression for both operands, or different expressions. For example, here we write the (fictitious) `'combine'` instruction with `bar` as its read-only source operand and `foo` as its read-write destination:

```
asm ("combine %2,%0" : "=r" (foo) : "0" (foo), "g" (bar));
```

The constraint `'"0"'` for operand 1 says that it must occupy the same location as operand 0. Therefore it is not necessary to substitute operand 1 into the assembler code output.

Usually the most convenient way to use these `asm` instructions is to encapsulate them in macros that look like functions. For example,

```
#define sin(x)       \
({ double __value, __arg = (x);   \
   asm ("fsinx %1,%0": "=f" (__value): "f" (__arg));  \
   __value; })
```

Here the variable `__arg` is used to make sure that the instruction operates on a proper `double` value, and to accept only those arguments `x` which can convert automatically to a `double`.

Another way to make sure the instruction operates on the correct data type is to use a cast in the `asm`. This is different from using a variable `__arg` in that it converts more different types. For example, if the desired type were `int`, casting the argument to `int` would accept a pointer with no complaint, while assigning the argument to an `int` variable named `__arg` would warn about using a pointer unless the caller explicitly casts it.

GNU C++ assumes for optimization purposes that these instructions have no side effects except to change the output operands. This does not mean that instructions with a side effect cannot be used, but you must be careful, because the compiler may eliminate them if the output operands aren't used, or move them out of loops, or replace two with one if they constitute a common subexpression. Also, if your instruction does have a side effect on a variable that otherwise appears not to change, the old value of the variable may be reused later if it happens to be found in a register.

You can prevent an `asm` instruction from being deleted, moved or combined by writing the keyword `volatile` after the `asm`. For example:

```
#define set_priority(x)  \
asm volatile ("set_priority %1":     \
              "=m" (*(char *)0): "g" (x))
```

Note that we have supplied an output operand which is not actually used in the instruction. This is because `asm` requires at least one output operand. This requirement exists for internal implementation reasons and we might be able to relax it in the future.

In this case output operand has the additional benefit effect of giving the appearance of writing in memory. As a result, GNU C++ will assume that data previously fetched from memory must be fetched again if needed again later. This may be desirable if you have not employed the `volatile` keyword on all the variable declarations that ought to have it.

6.15 Controlling Names Used in Assembler Code

You can specify the name to be used in the assembler code for a C++ function or variable (including static class variables and global anonymous union members) by writing the `asm` keyword after the declarator as follows:

```
int foo asm ("myfoo") = 2;
```

This specifies that the name to be used for the variable `foo` in the assembler code should be '`myfoo`' rather than the usual '`_foo`'.

On systems where an underscore is normally prepended to the name of a C++ function or variable, this feature allows you to define names for the linker that do not start with an underscore.

You cannot use `asm` in this way in a function *definition*; but you can get the same effect by writing a declaration for the function before its definition and putting `asm` there, like this:

```
extern func () asm ("FUNC");

func (int x, int y)
...
```

It is up to you to make sure that the assembler names you choose do not conflict with any other assembler symbols. Also, you must not use a register name; that would produce completely invalid assembler code. GNU C++ does not as yet have the ability to store static variables in registers. Perhaps that will be added.

The GNU C++ language makes some extensions to the C++ `new` operator, and adds a few non-C operators as well.

6.16 Non-C operators: `operator <?` and `operator >?`

It is very convenient to have operators which return the "minimum" or the "maximum" of two arguments. It is possible to specify a form to return the minimum of two things in C, as the following example shows.

```
#define MIN(X,Y) ((X) < (Y) ? : (X) : (Y))
```

However, this is not clean, because side-effects in `X` or `Y` may cause unintended behavior. It also requires that users employ a functional notation for a fundamental operation.

6.17 Controlling `operator new`.

The user now has much more control over the operator new. Normally, operator new calls the function `__builtin_new` with a `size` argument, and `__builtin_new` returns a pointer to a block of storage at least `size` bytes long. It is now possible to pass arguments to operator new, which will pass those arguments, along with a size parameter, to a function called `__user_new`, which can return anything the users wants it to. It is the user's responsibility to define `__user_new`. The function `__user_new` may be declared overloaded, just like any other function. It is not implicitly overloaded. The following is an example of its use:

```
enum mem_type { GLOBAL, LOCAL };

extern void *__user_new (int, mem_type);
extern void do_it (int*, int*);

void foo ()
{
  int *pi = new { LOCAL } int;
  int *ai = new { GLOBAL } int[50];

  do_it (pi, ai);
}
```

In this example, a storage allocator capable of allocating different types of memory can be utilized in a relatively clean way. All other semantics of 'new' are preserved: initializations are performed in exactly the same manner. One must be careful when using such a function, however: it is up to the constructors to know what actions to take if the memory returned from `__user_new` is not in their address space.

6.18 Function calls as first-class objects

In some cases it is desirable not to execute a function call immediately, but to perform some actions before the function is to be called, call the function, or otherwise obtain a value as a function of the called function's code and its arguments, perform some more actions afterwards, and return a result. An example of such a use would be the execution of remote procedure calls on a distributed system. One may get a request on one node to apply a function to some arguments, but that function may actually be on another node (as might some of the arguments). A "wrapper" allows a function call to be turned into an argument list which includes as arguments an encoding of the function being "wrapped" and its arguments. From this, it is possible, for example, to send a message to the node where the function actually lives, along with an encoding of the arguments

as a list, have that function execute, return its result via another message, and ultimately return a result to the caller. A "wrapper" allows the user great flexibility in the implementation of such behaviors, flexibility which allows one to specify many different ways of implementing the semantics of a function call, without requiring that the actual code for the function being wrapped be modified in any way.

An example will demonstrate one use of wrappers. This is a highly experimental feature, and one which should be expected to evolve suddenly. Users are therefore encouraged to participate in this evolutionary process.

```
// Memoizing example.  Normal C++ code.

class NumTheory
{
    // Use a hash table for memoizing
    HashTable h;

    int lookup (int (NumTheory::*)(int), int);
    int install (int (NumTheory::*)(int), int, int);

  public:
    // Some functions to memoize
    int fib (int);                      // Fibonacci numbers
    int prime (int);                    // Prime Numbers
}

int NumTheory::fib (int n)
{
    if (n == 0)
      return 0;
    if (n == 1)
      return 1;
    return fib (n - 1) + fib (n - 2);
}

main (int, char *[])
{
    NumTheory n;

    // find the 100th prime.
    int p100 = n.prime (100);
    // find the 101st prime--might be fast, since we know the 100th
    int p101 = n.prime (101);

    // find the 10th Fibonacci number
    int f10 = n.fib (10);
    // find the 11th Fibonacci number--might also be fast
    int f11 = n.fib (11);

    printf ("The 100th prime number is %d\n", p100);
    printf ("The 101st prime number is %d\n", p101);
    // etc ...
}
```

A "wrapper" can be added to the class declaration of 'NumTheory' with the following declaration:

```
()NumTheory (int (NumTheory::*)(int), int);
```

A wrapper declaration is syntactically valid anywhere a member function declaration is. By adding this declaration, the compiler catches out calls to member functions of the class 'NumTheory' and replaces them with calls to the wrapper. A wrapper may therefore be used to implement memoizing as follows:

```
// This wrapper uses previously computed results, if available.
// Newly generated results are entered into the hash table.
int NumTheory::()NumTheory (int (NumTheory::*pf)(int), int arg)
{
    // try to use a previously computed value.
    int val = hash (pf, arg);
    if (val < 0)
      {
        // Must compute value.
        val = (this->*pf)(arg);
        // Save it into hash table.
        install (pf, arg, val);
      }
    // else, we can use previously computed value.
    return val;
}
```

A paper is underway to more fully explain the possible uses and implementations of wrappers.

7. Features of GNU C++

Because the GNU C++ compiler is a native code compiler, there are opportunities for optimization which do not exist for C++ to C translators. Assitional features which have been ignored by other systems, but implemented by the GNU C++ system, are also listed here.

7.1 Structures as Arguments

The GNU C++ compiler can pass structures as arguments to functions. This can be difficult in C++, since the structure may take a constructor (the constructor of the form X(X&) in particular), in which case the structure passed must be initialized by that constructor as it lies on the stack. The GNU C++ compiler takes care of initializing the stack space before the function is called. If a destructor for the structure is defined, then GNU C++ also takes care of protecting that space until it is in fact destroyed.

When it comes to code generation, the advantage of this feature is that all structure parameters can be accessed via the same register: the frame pointer (or the stack pointer if the frame pointer can be eliminated). Other strategies which require passing pointers to structures either must use multiple indirections to access elements of the structures, or they must use extra registers to point to the structures.

7.2 The goto statement in GNU C++

The goto statement can be used to exit blocks which contain aggregates requiring destructors. The destructors are called, and then the goto is performed.

7.3 Type instantiation in GNU C++

Overloaded operators and user-defined type conversions provide the user with the freedom to implement abstract data types, often with very a natural syntax. This freedom places a certain burden on the compiler, however. Whenever an object which is of some user-defined type appears in a context where a built-in type must appear, the compiler must make the proper choices in translating the code from its C++ representation to something which executes in a predicable way. For example, in the following code:

```
struct A

  A ();
  int operator+ (A&);
  operator int ();
;

struct B

  B ();
  operator int ();
;

main ()

  A a;
  B b;

  int i = a + b;
  // some more stuff...
```

the compiler may try to overload `operator+`, but when that fails, it must be prepared to convert vaiables `a` and `b` to integer type, and perform normal addition. Sometimes it is not possible to determine, strictly from a bottom-up translation, what conversions should take place. Sometimes a technique known as type instantiation can help. Here is an expample:

```
struct A

  A ();
  A& operator *(); // The indirection operator
  A& operator *(A&); // The multiply operator
;

main ()

  A a;
  A& (*pf)(A*, A&);

  pf = a.operator*; // Which operator??
```

Type instantiation allows the compiler to choose from among many candidates, the most appropriate one given contextual information such as type. If an ambiguity still remains, then a warning or an error message is given.

7.4 Incremetal Linking and GNU C++

Incremetal linking adds a new dimension to programming with GNU C++. Standard C++ requires that constructors and destructors be called on all objects which declare them, including global data. For example, when using the '`stream`' library facilities, the global variables `cin`, `cout`, and `cerr` are global variables which are initialized before control ever reaches `main`. GNU C++ extends this model to incorporate code which is linked with a program at run-time. Code which illustrates this behavior is in the GNU C++ library, in the files '`test.hello.cc`', '`test.bye`', and others which comprise '`test0`', one of the tests in the '`Makefile`' in the GNU C++ library directory.

Care should be taken when writing code which uses incremental linking. Because GNU C++ can put more than just program code in text space (read-only variables and character strings typically into text space as well), the starting address of text space may not always be the first function defined. It is therefore suggested that users include the file '`Incremental.h`' as the *first* file included in any program which will be linked incrementally, and immediately use the macro `DECLARE_INIT_FUNCTION` to tell GNU C++ what function should be called to initialize the module.

8. Reporting Bugs

Your bug reports play an essential role in making GNU C++ reliable.

Reporting a bug may help you by bringing a solution to your problem, or it may not. But in any case the important function of a bug report is to help the entire community by making the next version of GNU C++ work better. Bug reports are your contribution to the maintenance of GNU C++.

In order for a bug report to serve its purpose, you must include the information that makes for fixing the bug.

8.1 Have You Found a Bug?

If you are not sure whether you have found a bug, here are some guidelines:

- If the compiler gets a fatal signal, for any input whatever, that is a compiler bug. Reliable compilers never crash.
- If the compiler produces invalid assembly code, for any input whatever (except an `asm` statement), that is a compiler bug, unless the compiler reports errors (not just warnings) which would ordinarily prevent the assembler from being run.
- If the compiler produces valid assembly code that does not correctly execute the input source code, that is a compiler bug.

  However, you must double-check to make sure, because you may have run into an incompatibility between GNU C++ and traditional C++/PCC (see chapter 5 [Incompatibilities], page 47). These incompatibilities might be considered bugs, but they are inescapable consequences of adding valuable features.

  Or you may have a program whose behavior is undefined, which happened by chance to give the desired results with another C++ compiler, or C++ front-end/C compiler combination.

  For example, in many nonoptimizing compilers, you can write '`x;`' at the end of a function instead of '`return x;`', with the same results. But the value of the function is undefined if '`return`' is omitted; it is not a bug when GNU C++ produces different results.

  Problems often result from expressions with two increment operators, as in '`f (*p++, *p++)`'. Your previous compiler might have interpreted that expression the way you intended; GNU C++ might interpret it another way; neither compiler is wrong.

  After you have localized the error to a single source line, it should be easy to check for these things. If your program is correct and well defined, you have found a compiler bug.

- If the compiler produces an error message for valid input, that is a compiler bug.
- If the compiler does not produce an error message for invalid input, that is a compiler bug. However, you should note that your idea of "invalid input" might be my idea of "an extension" or "support for traditional practice".
- If you are an experienced user of C++ compilers, your suggestions for improvement of GNU C++ are welcome in any case.

8.2 How to Report Bugs

Send bug reports for GNU C++ to one of these addresses:

```
bug-g++@prep.ai.mit.edu
{ucbvax|mit-eddie|uunet}!prep.ai.mit.edu!bug-g++
```

As a last resort, snail them to:

```
GNU C++ Compiler Bugs
c/o MCC
3500 West Balcones Center Drive
ATTN: Michael Tiemann
Austin, TX 78759
```

The fundamental principle of reporting bugs usefully is this: **report all the facts**. If you are not sure whether to mention a fact or leave it out, mention it!

Often people omit facts because they think they know what causes the problem and they conclude that some details don't matter. Thus, you might assume that the name of the variable you use in an example does not matter. Well, probably it doesn't, but one cannot be sure. Perhaps the bug is a stray memory reference which happens to fetch from the location where that name is stored in memory; perhaps, if the name were different, the contents of that location would fool the compiler into doing the right thing despite the bug. Play it safe and give an exact example.

If you want to enable me to fix the bug, you should include all these things:

- The version of GNU C++. You can get this by running it with the '-v' option.

  Without this, I won't know whether there is any point in looking for the bug in the current version of GNU C++.

- A complete input file that will reproduce the bug. If the bug is in the C preprocessor, send me a source file and any header files that it requires. If the bug is in the compiler proper ('`c++`'), run your source file through the C preprocessor by doing '`g++ -E` *sourcefile* `>` *outfile*', then include the contents of *outfile* in the bug report. (Any '`-I`', '`-D`' or '`-U`' options that you used in actual compilation should also be used when doing this.)

  A single statement is not enough of an example. In order to compile it, it must be embedded in a function definition; and the bug might depend on the details of how this is done.

  Without a real example I can compile, all I can do about your bug report is wish you luck. It would be futile to try to guess how to provoke the bug. For example, bugs in register allocation and reloading frequently depend on every little detail of the function in which they happen.

- The command arguments you gave GNU C++ to compile that example and observe the bug. For example, did you use '`-O`'? To guarantee you won't omit something important, list them all.

  If I were to try to guess the arguments, I would probably guess wrong and then I would not encounter the bug.

- The names of the files that you used for '`tm.h`' and '`md`' when you installed the compiler.

- The type of machine you are using, and the operating system name and version number.

- A description of what behavior you observe that you believe is incorrect. For example, "It gets a fatal signal," or, "There is an incorrect assembler instruction in the output."

  Of course, if the bug is that the compiler gets a fatal signal, then I will certainly notice it. But if the bug is incorrect output, I might not notice unless it is glaringly wrong. I won't study all the assembler code from a 50-line C program just on the off chance that it might be wrong.

  Even if the problem you experience is a fatal signal, you should still say so explicitly. Suppose something strange is going on, such as, your copy of the compiler is out of synch, or you have encountered a bug in the C library on your system. (This has happened!) Your copy might crash and mine would not. If you *told* me to expect a crash, then when mine fails to crash, I would know that the bug was not happening for me. If you had not told me to expect a crash, then I would not be able to draw any conclusion from my observations.

  In cases where GNU C++ generates incorrect code, if you send me a small complete sample program I will find the error myself by running the program under a debugger. If you send me a large example or a part of a larger program, I cannot do this; you must debug the compiled program and narrow the problem down to one source line. Tell me which source line it is, and what you believe is incorrect about the code generated for that line.

- If you send me examples of output from GNU C++, please use '`-g`' when you make them. The debugging information includes source line numbers which are essential for correlating the output with the input.

Here are some things that are not necessary:

- A description of the envelope of the bug.

  Often people who encounter a bug spend a lot of time investigating which changes to the input file will make the bug go away and which changes will not affect it.

  This is often time consuming and not very useful, because the way I will find the bug is by running a single example under the debugger with breakpoints, not by pure deduction from a series of examples.

  Of course, it can't hurt if you can find a simpler example that triggers the same bug. Errors in the output will be easier to spot, running under the debugger will take less time, etc. An easy way to simplify an example is to delete all the function definitions except the one where the bug occurs. Those earlier in the file may be replaced by external declarations.

  However, simplification is not necessary; if you don't want to do this, report the bug anyway.

- A patch for the bug.

  A patch for the bug does help me if it is a good one. But don't omit the necessary information, such as the test case, because I might see problems with your patch and decide to fix the problem another way.

  Sometimes with a program as complicated as GNU C++ it is very hard to construct an example that will make the program go through a certain point in the code. If you don't send me the example, I won't be able to verify that the bug is fixed.

- A guess about what the bug is or what it depends on.

  Such guesses are usually wrong. Even I can't guess right about such things without using the debugger to find the facts. They also don't serve a useful purpose.

9. GNU C++ and Portability

The main goal of GNU C++ was to make a good, fast compiler for machines in the class that the GNU system aims to run on: 32-bit machines that address 8-bit bytes and have several general registers. Elegance, theoretical power and simplicity are only secondary.

GNU C++ gets most of the information about the target machine from a machine description which gives an algebraic formula for each of the machine's instructions. This is a very clean way to describe the target. But when the compiler needs information that is difficult to express in this fashion, I have not hesitated to define an ad-hoc parameter to the machine description. The purpose of portability is to reduce the total work needed on the compiler; it was not of interest for its own sake.

GNU C++ does not contain machine dependent code, but it does contain code that depends on machine parameters such as endianness (whether the most significant byte has the highest or lowest address of the bytes in a word) and the availability of auto-increment addressing. In the RTL-generation pass, it is often necessary to have multiple strategies for generating code for a particular kind of syntax tree, strategies that are usable for different combinations of parameters. Often I have not tried to address all possible cases, but only the common ones or only the ones that I have encountered. As a result, a new target may require additional strategies. You will know if this happens because the compiler will call `abort`. Fortunately, the new strategies can be added in a machine-independent fashion, and will affect only the target machines that need them.

The implementation of pointers to virtual member functions is not entirely portable. This is because to be truly portable, these pointers would have to be twice the size of normal pointers. The assumption that is made is that offsets into a virtual function table can be distinguished from addresses of functions. In GNU C++, there are two ways of doing this: if the assumption is made that the largest offset into a virtual function table will always be smaller than the first text address available to the user, then define the symbol `VTABLE_USES_MASK`, and set `VINDEX_MAX` to the largest power of two less than or equal to that size. When GNU C++ programs are linked with the GNU linker and `crt0+.o`, the GNU C++ startup code, a check is performed that virtual tables did not exceed this size when the program is run.

If `VTABLE_USES_MASK` is not defined, then the compiler assumes that pointers with their high bit set are offsets into the virtual function table, otherwise they are pointers to addresses in text space. Neither one of these strategies is particularly attractive for machines with segmented architectures with small segments, but then again, for these machines, nothing is.

Pointers to static class members are not implemented. It was felt that use of this feature would

be extremely rare, and the run-time overhead associated with the implementation of this feature would, in general, not be worth it.

10. Interfacing to GNU C++ Output

GNU C++ is normally configured to use the same function calling convention normally in use on the target system. This is done with the machine-description macros described (See the `Machine Macros` section of the "Internals of GNU CC" document).

However, returning of structure and union values is done differently. As a result, functions compiled with PCC returning such types cannot be called from code compiled with GNU C++, and vice versa. This usually does not cause trouble because the Unix library routines don't return structures and unions.

Structures and unions that are 1, 2, 4 or 8 bytes long are returned in the same registers used for `int` or `double` return values. (GNU C++ typically allocates variables of such types in registers also.) Structures and unions of other sizes are returned by storing them into an address passed by the caller in a register. This method is faster than the one normally used by PCC and is also reentrant. The register used for passing the address is specified by the machine-description macro `STRUCT_VALUE_REGNUM`.

GNU C++ always passes arguments on the stack. At some point it will be extended to pass arguments in registers, for machines which use that as the standard calling convention. This will make it possible to use such a convention on other machines as well. However, that would render it completely incompatible with PCC. We will probably do this once we have a complete GNU system so we can compile the libraries with GNU C++.

If you use `longjmp`, beware of automatic variables. ANSI C says that automatic variables that are not declared `volatile` have undefined values after a `longjmp`. And this is all GNU C++ promises to do, because it is very difficult to restore register variables correctly, and one of GNU C++'s features is that it can put variables in registers without your asking it to.

If you want a variable to be unaltered by `longjmp`, and you don't want to write `volatile` because old C compilers don't accept it, just take the address of the variable. If a variable's address is ever taken, even if just to compute it and ignore it, then the variable cannot go in a register:

```
{
  int careful;
  &careful;
  ...
}
```

Code compiled with GNU C++ may call certain library routines. The routines needed on the Vax and 68000 are in the file '`gnulib.c`'. You must compile this file with the standard C compiler, not with GNU C++, and then link it with each program you compile with GNU C++. The usual function call interface is used for calling the library routines. Some standard parts of the C library, such as `bcopy`, are also called automatically.

The file '`gnulib.c`' also provides the implementation for the functions `__builtin_new` and `__builtin_delete`, the functions responsible for actually allocating and deallocating storage for GNU C++ programs.

11. Passes and Files of the Compiler

The overall control structure of the compiler is in '`toplev.c`'. This file is responsible for initialization, decoding arguments, opening and closing files, and sequencing the passes. For information about the internals of the GNU C++ compiler, at this level and below it is functionally identical to the GNU CC compiler. Please consult that document for further details.

The parsing pass is invoked only once, to parse the entire input. The RTL intermediate code for a function is generated as the function is parsed, a statement at a time. Each statement is read in as a syntax tree and then converted to RTL; then the storage for the tree for the statement is reclaimed. Storage for types (and the expressions for their sizes), declarations, and a representation of the binding contours and how they nest, remains until the function is finished being compiled; these are all needed to output the debugging information.

Each time the parsing pass reads a complete function definition or top-level declaration, it calls the function `rest_of_compilation` or `rest_of_decl_compilation` in '`toplev.c`', which are responsible for all further processing necessary, ending with output of the assembler language. All other compiler passes run, in sequence, within `rest_of_compilation`. When that function returns from compiling a function definition, the storage used for that function definition's compilation is entirely freed, unless it is an inline function.

Here is a list of all the passes of the compiler and their source files. Also included is a description of where debugging dumps can be requested with '`-d`' options.

- Parsing. This pass reads the entire text of a function definition, constructing partial syntax trees. This and RTL generation are no longer truly separate passes (formerly they were), but it is easier to think of them as separate.

  The tree representation does not entirely follow C++ syntax, because it is intended to support other languages as well.

  C++ data type analysis is also done in this pass, and every tree node that represents an expression has a data type attached. Variables are represented as declaration nodes.

  Constant folding and associative-law simplifications are also done during this pass.

  The source files for parsing are '`parse.y`', '`decl.c`', '`typecheck.c`', '`stor-layout.c`', '`fold-const.c`', and '`tree.c`'. The last three are intended to be language-independent. There are also header files '`parse.h`', '`c-tree.h`', '`tree.h`' and '`tree.def`'. The last two define the format of the tree representation.
- RTL generation. This is the conversion of syntax tree into RTL code. It is actually done statement-by-statement during parsing, but for most purposes it can be thought of as a separate

pass. Constructors and destructors are processed specially by `finish_function`.

This is where the bulk of target-parameter-dependent code is found, since often it is necessary for strategies to apply only when certain standard kinds of instructions are available. The purpose of named instruction patterns is to provide this information to the RTL generation pass.

Optimization is done in this pass for `if`-conditions that are comparisons, boolean operations or conditional expressions. Tail recursion is detected at this time also. Decisions are made about how best to arrange loops and how to output `switch` statements.

The source files for RTL generation are '`stmt.c`', '`expr.c`', '`explow.c`', '`expmed.c`', '`optabs.c`' and '`emit-rtl.c`'. Also, the file '`insn-emit.c`', generated from the machine description by the program `genemit`, is used in this pass. The header files '`expr.h`' is used for communication within this pass.

The header files '`insn-flags.h`' and '`insn-codes.h`', generated from the machine description by the programs `genflags` and `gencodes`, tell this pass which standard names are available for use and which patterns correspond to them.

Aside from debugging information output, none of the following passes refers to the tree structure representation of the function (only part of which is saved).

The decision of whether the function can and should be expanded inline in its subsequent callers is made at the end of rtl generation. The function must meet certain criteria, currently related to the size of the function and the types and number of parameters it has. Note that this function may contain loops, recursive calls to itself (tail-recursive functions can be inlined!), gotos, in short, all constructs supported by GNU CC.

The option '`-dr`' causes a debugging dump of the RTL code after this pass. This dump file's name is made by appending '`.rtl`' to the input file name.

- Jump optimization. This pass simplifies jumps to the following instruction, jumps across jumps, and jumps to jumps. It deletes unreferenced labels and unreachable code, except that unreachable code that contains a loop is not recognized as unreachable in this pass. (Such loops are deleted later in the basic block analysis.)

  Jump optimization is performed two or three times. The first time is immediately following RTL generation. The second time is after CSE, but only if CSE says repeated jump optimization is needed. The last time is right before the final pass. That time, cross-jumping and deletion of no-op move instructions are done together with the optimizations described above.

  The source file of this pass is '`jump.c`'.

  The option '`-dj`' causes a debugging dump of the RTL code after this pass is run for the first time. This dump file's name is made by appending '`.jump`' to the input file name.

- Register scan. This pass finds the first and last use of each register, as a guide for common subexpression elimination. Its source is in '`regclass.c`'.

- Common subexpression elimination. This pass also does constant propagation. Its source file

is 'cse.c'. If constant propagation causes conditional jumps to become unconditional or to become no-ops, jump optimization is run again when CSE is finished.

The option '-ds' causes a debugging dump of the RTL code after this pass. This dump file's name is made by appending '.cse' to the input file name.

- Loop optimization. This pass moves constant expressions out of loops. Its source file is 'loop.c'.

  The option '-dL' causes a debugging dump of the RTL code after this pass. This dump file's name is made by appending '.loop' to the input file name.

- Stupid register allocation is performed at this point in a nonoptimizing compilation. It does a little data flow analysis as well. When stupid register allocation is in use, the next pass executed is the reloading pass; the others in between are skipped. The source file is 'stupid.c'.

- Data flow analysis ('flow.c'). This pass divides the program into basic blocks (and in the process deletes unreachable loops); then it computes which pseudo-registers are live at each point in the program, and makes the first instruction that uses a value point at the instruction that computed the value.

  This pass also deletes computations whose results are never used, and combines memory references with add or subtract instructions to make autoincrement or autodecrement addressing.

  The option '-df' causes a debugging dump of the RTL code after this pass. This dump file's name is made by appending '.flow' to the input file name. If stupid register allocation is in use, this dump file reflects the full results of such allocation.

- Instruction combination ('combine.c'). This pass attempts to combine groups of two or three instructions that are related by data flow into single instructions. It combines the RTL expressions for the instructions by substitution, simplifies the result using algebra, and then attempts to match the result against the machine description.

  The option '-dc' causes a debugging dump of the RTL code after this pass. This dump file's name is made by appending '.combine' to the input file name.

- Register class preferencing. The RTL code is scanned to find out which register class is best for each pseudo register. The source file is 'regclass.c'.

- Local register allocation ('local-alloc.c'). This pass allocates hard registers to pseudo registers that are used only within one basic block. Because the basic block is linear, it can use fast and powerful techniques to do a very good job.

  The option '-dl' causes a debugging dump of the RTL code after this pass. This dump file's name is made by appending '.lreg' to the input file name.

- Global register allocation ('global-alloc.c'). This pass allocates hard registers for the remaining pseudo registers (those whose life spans are not contained in one basic block).

- Reloading. This pass renumbers pseudo registers with the hardware registers numbers they were allocated. Pseudo registers that did not get hard registers are replaced with stack slots. Then it finds instructions that are invalid because a value has failed to end up in a register,

or has ended up in a register of the wrong kind. It fixes up these instructions by reloading the problematical values temporarily into registers. Additional instructions are generated to do the copying.

Source files are '`reload.c`' and '`reload1.c`', plus the header '`reload.h`' used for communication between them.

The option '`-dg`' causes a debugging dump of the RTL code after this pass. This dump file's name is made by appending '`.greg`' to the input file name.

- Jump optimization is repeated, this time including cross-jumping and deletion of no-op move instructions. Machine-specific peephole optimizations are performed at the same time.

  The option '`-dJ`' causes a debugging dump of the RTL code after this pass. This dump file's name is made by appending '`.jump2`' to the input file name.

- Final. This pass outputs the assembler code for the function. It is also responsible for identifying spurious test and compare instructions. The function entry and exit sequences are generated directly as assembler code in this pass; they never exist as RTL.

  The source files are '`final.c`' plus '`insn-output.c`'; the latter is generated automatically from the machine description by the tool '`genoutput`'. The header file '`conditions.h`' is used for communication between these files.

- Debugging information output. This is run after final because it must output the stack slot offsets for pseudo registers that did not get hard registers. Source files are '`dbxout.c`' for DBX symbol table format and '`symout.c`' for GDB's own symbol table format.

Some additional files are used by all or many passes:

- Every pass uses '`machmode.def`', which defines the machine modes.
- All the passes that work with RTL use the header files '`rtl.h`' and '`rtl.def`', and subroutines in file '`rtl.c`'. The tools `gen*` also use these files to read and work with the machine description RTL.
- Several passes refer to the header file '`insn-config.h`' which contains a few parameters (C macro definitions) generated automatically from the machine description RTL by the tool `genconfig`.
- Several passes use the instruction recognizer, which consists of '`recog.c`' and '`recog.h`', plus the files '`insn-recog.c`' and '`insn-extract.c`' that are generated automatically from the machine description by the tools '`genrecog`' and '`genextract`'.
- Several passes use the header files '`regs.h`' which defines the information recorded about pseudo register usage, and '`basic-block.h`' which defines the information recorded about basic blocks.
- '`hard-reg-set.h`' defines the type `HARD_REG_SET`, a bit-vector with a bit for each hard register, and some macros to manipulate it. This type is just `int` if the machine has few enough hard

registers; otherwise it is an array of `int` and some of the macros expand into loops.

12. The Configuration File

The configuration file '`config-`*machine*`.h`' contains macro definitions that describe the machine and system on which the compiler is running. Most of the values in it are actually the same on all machines that GNU CC runs on, so most all configuration files are identical. But there are some macros that vary:

`FAILURE_EXIT_CODE`

A C expression for the status code to be returned when the compiler exits after serious errors.

`SORRY_EXIT_CODE`

A C expression for the status code to be returned when the compiler exits after compiling a file which used a feature not yet implemented.

`SUCCESS_EXIT_CODE`

A C expression for the status code to be returned when the compiler exits without serious errors.

13. Things still left to do for GNU C++

The GNU C++ grammar is an LALR grammar in Bison format. It currently uses the simple LALR parser driver (`bison.simple`). It would be hard, but not impossible, to adapt GNU C++ to take full advantage of the Bison parsing machinery (`bison.hairy`), so that syntactic ambiguities which led to semantic errors could be unparsed, and reparsed with different syn-tactics. This would give the GNU C++ parser the same heuristic power as a recursive descent parser, while maintaining an LALR grammar basis.

Applications which make heavy use of virtual functions can pay a high price for function call overhead to its virtual functions. Small virtual functions are particularly troublesome because call overhead is high, and they cannot usually be inlined to take care of that. A more efficient calling sequence, which preserves both the class variable (`this`) and its virtual function table pointer, could eliminate memory traffic in many cases for these two often used parameters.

14. List of currently known bugs in GNU C++

The single greatest mis-feature of GNU C++ is that it cannot handle C-style function definitions. It also does not handle pointer to function declarations gracefully in a number of contexts, especially in parameter declarations.

GNU C++ does not yet handle local class declarations. I.e., a local class declaration will permanently shadow a previous declaration. It also cannot handle class declarations which contain inline function definitions within the scope of a function.

Constructors within class declarations cannot be declared "inline". In fact, they cannot be declared with any leading storage class specifiers or type specifiers in that syntactic position. To declare a constructor inline, define the constructor outside of the class declaration, i.e., `inline X::X(...)`.

The '`-p`' and '`-pg`' options are not yet supported. The reason for this is that to work with these options, a special crt0.o file must be used. Such files have not yet been modified to work with GNU C++.

GNU C++ does not appear to conform to the visibility rules to which the AT&T compiler does. When these rules are better defined (or better explained), GNU C++ will implement them properly.

15. Related documentation and bibliography

[1] Stroustrup, Bjarne: *The C++ Programming Language.* Addison-Wesley, 1986.

[2] Stroustrup, Bjarne: *The Evolution of C++: 1985 to 1987* First USENIX C++ Workshop Proceedings, 1987.

[3] Stallman, Richard: *The Internals of GNU CC* Free Software Foundation, 1988.

[4] Stallman, Richard: *The GNU Debugger for GNU C++* Free Software Foundation, 1988.

[5] Stallman, Richard: *GNU Emacs Manual* Fifth Edition, Emacs Version 18 for Unix Users, Free Software Foundation, October 1986.

[6] Tiemann, Michael: *Open Issues in G++ Language Development* MCC, Technical Report, *in progress.*